Planification Optimiste pour Systèmes Déterministes

Jean-Francois Hren

École doctorale Sciences Pour l'Ingénieur

T H È S E

présentée en vue d'obtenir le grade de

Docteur, spécialité Informatique

par

Jean-François Hren

Planification Optimiste pour Systèmes Déterministes

préparée dans l'équipe-projet SequeL commune

rédigée en français,

soutenue en français le 21 Juin 2012 devant le jury composé de

Rapporteurs :

Bruno Bouzy, Maître de Conférences	-	Université Paris Descartes
Brahim Chaib-Draa, Professeur	-	Université Laval

Examinateurs :

Damien Ernst, Associate Professor	-	Université de Liège
Olivier Pietquin, Directeur de Recherche	-	CNRS
Bruno Scherrer, Research Scientist (CR1)	-	INRIA

Directeur :

Rémi Munos, Senior Researcher (DR2)	-	INRIA

Remerciements

Je remercie mon directeur de thèse, Rémi Munos, pour la patience et la confiance qu'il m'a témoigné (particulièrement à chaque réinscription administrative à l'université) et ce pendant ces 5 années. Je remercie Bruno Bouzy et Brahim Chaib-Draa qui ont gentiment accepté d'être rapporteurs de ma thèse. Je remercie aussi Damien Ernst, Olivier Pietquin et Bruno Scherrer d'avoir accepté d'être examinateurs pendant ma soutenance de thèse.

Je remercie ma famille de m'avoir encourager à continuer mes études (ou du moins, à ne pas m'avoir décourager). Je remercie Carole pour son aide et l'ajout après sa relecture éclairée d'un nombre substantiel de virgules. Je remercie les membres passés et présents du bureau 11bis pour m'avoir supporter (dans les 2 sens du termes) : Alexandra, Azal, Hachem, Manu, Olivier et Victor. Je remercie les membre de l'équipe SequeL qui m'ont fourni un environnement propice à la recherche. Je remercie les membres passés et présents de l'équipe POPS/FUN/2XS (entourez la bonne réponse) pour leur bonne humeur durant nos nombreux repas du midi. Je remercie l'équipe pédagogie du département Informatique de l'IUT A de Lille 1 pour m'avoir accordé leur confiance pendant mes années de monitorat et d'ATER sans lesquelles ma thèse n'aurait pu se poursuivre. Enfin je remercie tous ceux qui de près ou de loin ont fait en sorte que cette thèse voit le jour.

Table des matières

Chapitre 1

Introduction

Dans le domaine de l'apprentissage par renforcement, la planification dans les processus de décisions markoviens est une approche en ligne utilisée pour contrôler un système dont on possède un modèle génératif. Nous nous proposons d'adresser ce problème dans le cas déterministe avec espace d'action discret ou continu.

Cette thèse s'attache au chapitre 2 à présenter succinctement les processus de décision markoviens puis l'apprentissage par renforcement. Nous présentons en particulier trois algorithmes centraux que sont l'itération de la valeur, l'itération de la politique et le Q-Learning.

Au chapitre 3, nous expliquons l'approche de la planification dans les processus de décision markoviens pour contrôler des systèmes en ligne. Ainsi, nous supposons posséder un modèle génératif d'un système à contrôler et nous l'utilisons pour décider, à chaque pas de temps du système à contrôler, de l'action à lui appliquer en vue de le faire transiter dans un état maximisant la somme future des récompenses dépréciées. Nous considérons un modèle génératif comme une boite noire, laquelle étant donnée un état et une action, nous retourne un état successeur ainsi qu'une récompense associée.

L'approche optimiste est détaillée dans sa philosophie et dans son application à la résolution du dilemme exploration-exploitation au travers de différentes techniques présentes dans la littérature.

Nous présentons différents algorithmes issus de la littérature et s'appliquant dans le cadre de la planification dans les processus de décision markoviens. Nous nous concentrons en particulier sur les algorithmes effectuant une recherche avant par construction d'un arbre des possibilités — *look-ahead tree* en anglais. Les algorithmes sont présentés et mis en relation les uns avec les autres. L'algorithme de recherche du plus court chemin dans un graphe A* est présenté en vue d'être relié à notre première contribution, l'algorithme de planification optimiste.

Nous détaillons cette première contribution au chapitre 4. Dans un premier temps, nous présentons en détail le contexte de la planification sous con-

trainte de ressources computationnelles ainsi que la notion de regret. Dans un second temps, l'algorithme de planification uniforme est présenté et son regret est analysé pour obtenir une base comparative avec l'algorithme de planification optimiste. Enfin, celui-ci est présenté et son regret est analysé. L'analyse est étendue à une classe de problèmes définie par la proportion de chemins ϵ-optimaux, permettant ainsi d'établir une borne supérieure sur le regret de l'algorithme de planification optimiste meilleure que celle de l'algorithme de planification uniforme dans le pire des cas.

Des expérimentations sont menées pour valider la théorie et chiffrer les performances de l'algorithme de planification optimiste par le biais de problèmes issus de la littérature comme le cart-pole, l'acrobot ou le mountain car et en comparaison à l'algorithme de planification uniforme, à l'algorithme UCT ainsi qu'à l'algorithme de recherche aléatoire. Nous verrons que, comme suggéré par la définition de la borne supérieure sur son regret, l'algorithme de planification optimiste est sensible au facteur de branchement ce qui nous mène à envisager le cas où l'espace d'action est continu. Ceci fait l'objet de nos deux autres contributions au chapitre 5.

Notre deuxième contribution est l'algorithme de planification lipschitzienne reposant sur une hypothèse de régularité sur les récompenses menant à supposer que la fonction de transition et la fonction récompense du processus de décision markovien modélisant le système à contrôler sont lipschitziennes. De cette hypothèse, nous formulons une borne sur un sous-ensemble de sous-espaces de l'espace d'action continu nous permettant de l'explorer par discrétisations successives. L'algorithme demande cependant la connaissance de la constante de Lipschitz associée au système à contrôler.

Des expérimentations sont menées pour évaluer l'approche utilisée pour différentes constantes de Lipschitz sur des problèmes de la littérature comme le cart-pole, l'acrobot ou la lévitation magnétique d'une boule en acier. Les résultats montrent que l'estimation de la constante de Lipschitz est difficile et ne permet pas de prendre en compte le paysage local des récompenses.

Notre troisième contribution est l'algorithme de planification séquentielle découlant d'une approche intuitive où une séquence d'instances d'un algorithme d'optimisation globale est utilisée pour construire des séquences d'actions issues de l'espace d'action continu.

Des expérimentations sont menées pour évaluer cet approche intuitive pour différents algorithmes d'optimisation globale sur des problèmes de la littérature comme le cart-pole, le bateau ou le nageur. Les résultats obtenus sont encourageants et valident l'approche intuitive.

Finalement, nous concluons en résumant les différentes contributions et en ouvrant sur de nouvelles perspectives et extensions.

Chapitre 2

Processus de Décision Markovien et Apprentissage par Renforcement

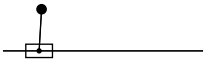
Dans le domaine de l'intelligence artificielle et, plus particulièrement celui de l'apprentissage par renforcement, un processus de décision markovien (MDP) ([Putterman, 1994], [Boutilier et al., 1999]) est utilisé pour modéliser un système à étudier. Ce système transite dans différents états par l'intermédiaire d'actions effectuées par un agent. Dans le cadre de l'apprentissage par renforcement, l'objectif de cet agent est de maximiser une somme des récompenses — dépréciées dans le temps ou non — qu'il obtiendra au grès de ses prises de décision dans ledit système. L'agent aura un comportement dit *optimal* — et donc désirable — si la somme des récompenses induite par ce comportant est aussi optimale.

De cette description sommaire, 4 éléments constituant un MDP peuvent être dégagés :

Les états représentent le système à un instant donné. Suivant le principe markovien, un état doit contenir assez d'informations pour permettre l'évaluation d'un état successeur. L'ensemble des états peut être fini ou infini, discret ou continu. Si le système modélisé était, par exemple, une voiture, l'état contiendrait entre autres la vitesse de la voiture, sa position ou bien encore l'usure des pneumatiques.

Les actions définissent les interactions possibles entre un agent et le système. L'ensemble des actions peut être discret ou continu. Pour reprendre l'exemple de la voiture, accélérer ou freiner sont toutes deux des actions possibles.

Les probabilités de transition définissent la dynamique du système. Étant donné un état et une action, les probabilités de transition nous donnent la probabilité de transiter dans un certain état résultant. Ainsi,



pour l'exemple de la voiture, les probabilités de transition définissent comme plus probable la décélération de la voiture si celle-ci a une vitesse non-nulle quand l'action « freiner » est effectuée. Par contre, effectuer l'action « freiner » n'aura aucune incidence sur la vitesse de la voiture si celle-ci est nulle.

La fonction récompense rétribue l'agent pour sa prise de décision dans un état donné. Cette récompense servira à guider l'apprentissage d'un comportement optimal par le biais de la somme des récompenses obtenues lors d'une séquence de prises de décisions.

Plus formellement, un MDP est un quadruplet (X, A, P, r) où X est l'ensemble des états par lesquels le système peut se passer, A est l'ensemble des actions à disposition de l'agent, P représente les probabilités de transition c'est-à-dire $P(x_{t+1}|x_t, a_t)$ est la probabilité de transiter dans un état $x_{t+1} \in X$ si l'action $a_t \in A$ est effectuée dans l'état $x_t \in X$ avec $t \geq 0$ et $r : X \times A \to \mathbb{R}$ est la fonction récompense. L'indice t dénote les pas de temps faisant évoluer le système. Nous supposerons que les MDP considérés sont stationnaires ; c'est-à-dire que P et r ne dépendent pas de t. Un MDP est dit épisodique s'il possède un ou plusieurs états terminaux. Un état x est dit terminal — ou absorbant — si $P(x|x, a) = 1, \forall a \in A$ et donc que $P(x'|x, a) = 0, \forall x' \in X \backslash x$ et $a \in A$. Nous supposerons aussi que les récompenses sont dépréciées dans le temps par un facteur de dépréciation $\gamma \in [0, 1[$. Ce facteur permet de donner plus ou moins d'importance aux récompenses futures, elles sont dépréciées. Ainsi, si le facteur est proche de 0, la somme des récompenses dépréciées serait en grande partie composée de la récompense immédiate. Dans le cas contraire, la somme des récompenses dépréciées tendrait vers une simple somme des récompenses donnant autant d'importance aux récompenses proches qu'aux récompenses futures.

La figure 2.1 de la page 12 illustre un MDP décrivant un jeu de dés épisodique. Le jeu consiste à obtenir le meilleur score possible avec deux jets maximum d'un dé à six faces. Il y a deux actions possibles a et a' correspondant respectivement à "Lancer le dé" et à "Ne rien faire". Le joueur lance le dé une première fois et observe son score actuel pouvant être 1, 2, 3, 4, 5 ou 6 correspondant respectivement aux états $(1, 1)$, $(1, 2)$, $(1, 3)$, $(1, 4)$, $(1, 5)$ et $(1, 6)$. Ensuite le joueur décide de lancer le dé — action a — ou de ne rien faire — action a'. Enfin une dernière action a' doit être entreprise pour obtenir le score final du jeu. C'est pendant cette transition que les seules récompenses non-nulles du MDP sont présentes. Le but est donc de savoir si un deuxième jet doit être effectué suivant le score du premier pour maximiser les chances de faire plus. Intuitivement, nous pouvons deviner que l'action optimale dans l'état $(1, 1)$ est a. En effet, dans le pire des cas nous ne pouvons qu'obtenir le même score. Le jeu étant épisodique, il nous faut au moins un état initial et un état terminal. L'état initial est $(0, 0)$ correspondant au début de la partie lorsqu'aucun jet de dès n'a été effectué,

le score est donc de 0. De même il y a six états terminaux $(-,1)$, $(-,2)$, $(-,3)$, $(-,4)$, $(-,5)$ et $(-,6)$ contenant chacun le score final du jeu.

Les probabilités de transition sont les suivantes :

$$\begin{aligned} P((1,n)|(0,0),a) &= \frac{1}{6} \text{ avec } n \in \{1,\ldots,6\} \\ P((2,n)|(1,n'),a) &= \frac{1}{6} \text{ avec } n,n' \in \{1\ldots K\} \\ P((2,n)|(1,n),a') &= 1 \text{ avec } n \in \{1,\ldots,6\} \\ P((-,n)|(2,n),a') &= 1 \text{ avec } n \in \{1,\ldots,6\} \end{aligned}$$

Les probabilités de transition non spécifiées sont nulles. Nous pouvons observer que le dé ici est considéré non truqué ; chaque score a la même probabilité d'être obtenu. La fonction récompense est définie ainsi :

$$r(x_t,a_t) = \begin{cases} n & \text{si } x_t = (2,n) \text{ et } a_t = a' \text{ avec } n \in \{1,\ldots,6\} \\ 0 & \text{sinon} \end{cases}$$

Ainsi seul l'avant dernier état de chaque épisode retourne une récompense non nulle.

Le but d'un agent est de maximiser la somme des récompenses dépréciées obtenue lors d'une séquence de prises de décisions. Les prises de décisions seront effectuées grâce à une politique π qui, à chaque état $x \in X$, associe une action $a \in A$ ou une distribution de probabilités sur A dans le cas d'une politique stochastique. Il est à noter que seules les politiques stationnaires seront considérées. En effet, étant donné la classe de MDP considérée ici, il existe une politique stationnaire déterministe optimale. Déterminer la politique optimale π^* permettant de maximiser la somme des récompenses dépréciées et ainsi obtenir un comportement optimal de l'agent sur le système est l'objectif central de la *programmation dynamique* mise au jour par [Bellman, 1957].

Avant de décrire le principe de la programmation dynamique, il convient de définir quelques concepts.

Le premier est la *fonction valeur* permettant de mesurer l'utilité d'un état, c'est-à-dire la somme des récompenses maximale qu'une séquence de prises de décisions ayant cet état comme état initial pourrait obtenir en moyenne. Si les séquences de prises de décisions suivent une politique fixée π, nous obtenons alors la fonction valeur d'une politique π permettant de l'évaluer. Soit la fonction valeur V^π qui, pour un état $x \in X$ et, étant donné une politique stationnaire déterministe $\pi \in \Pi_{\text{stat}-\text{det}}$, retourne la somme des récompenses dépréciées le long des trajectoires démarrant de x et suivant la politique π :

$$V^\pi(x) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(x_t, \pi(x_t)) | x_0 = x \right]. \tag{2.1}$$

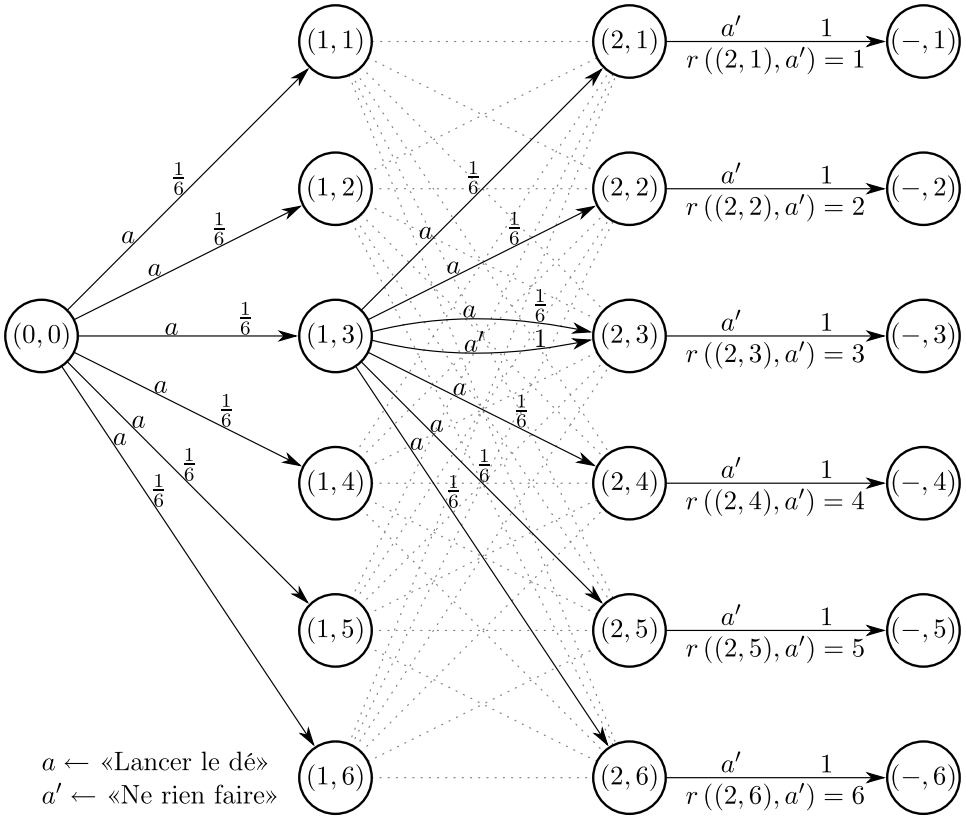


FIGURE 2.1 – Exemple d'un jeu de dés modélisé par un MDP. Une partie des transitions a été mis en pointillés pour améliorer la lisibilité.

$\pi(x_t)$ retourne l'action a_t associée à l'état x_t par la politique π. Cette action sera ensuite effectuée en l'état x_t et le système transitera en l'état $x_{t+1} \sim P(.|x_t, a_t)$. $V^\pi(x)$ est donc l'espérance sur la somme des récompenses dépréciées des séquences infinies de prises de décisions suivant la politique π et ayant comme état initial x.

De la même manière, nous pouvons définir la *fonction qualité* Q^π pour un état $x_0 \in X$ et une action initiale $a \in A$ étant donné une politique stationnaire déterministe $\pi \in \Pi_{\text{stat}-\text{det}}$ tel que :

$$Q^\pi(x, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(x_t, \pi(x_t)) | x_0 = x, a_0 = a \right]. \tag{2.2}$$

Les équations (2.1) et (2.2) sont également vraies si la politique stationnaire π est stochastique. Dans ce cas là, les actions a_t avec $t \geq 0$ seront tirées suivant $\pi(.|x_t)$.

Une fonction valeur V^* est dite optimale si elle vérifie :

$$V^*(x) = \max_{\pi \in \Pi_{\text{stat}}} V^\pi(x), \forall x \in X.$$

V^* contient la somme des récompenses dépréciées maximale pour tous les états $x \in X$ et comme dit précédemment, il existe une politique π^* qui pour tout état $x \in X$ vérifie $V^*(x) = V^{\pi^*}(x)$. Il est cependant difficile d'itérer sur toutes les politiques possibles pour trouver celles remplissant ces critères surtout si la taille de l'espace d'état n'est pas négligeable. Ainsi, chercher à calculer V^* sans passer par cette itération est un des résultats de la programmation dynamique. Elle permet de calculer de manière efficace V^* et Q^* pour ensuite pouvoir générer une politique optimale.

La programmation dynamique repose sur le principe d'optimalité de Bellman, établissant qu'une politique est optimale si, pour tout état, la première prise de décision et la séquence de prises de décision découlant de l'état résultant de la première prise de décision, sont optimales. En d'autres termes, il revient à décomposer le problème de prise de décisions en sous-problèmes de difficulté moindre. De ce principe, découle les équations d'optimalités de Bellman dont voici celle pour V^* — on considérera ici le cas où l'espace d'état X et l'espace d'action A sont tous deux finis :

$$V^*(x) = \max_{a \in A} \left\{ r(x, a) + \gamma \sum_{x' \in X} P(x'|x, a) V^*(x') \right\}. \tag{2.3}$$

Ainsi, l'optimalité de $V^*(x)$ est dépendante de la première prise de décision mais aussi, de l'optimalité de tous ses états successeurs possibles — états appartenant à $x' \in X | P(x'|x, a) \neq 0$.

Nous pouvons aussi définir V^* à partir de Q^* mais l'inverse est vrai aussi :

$$V^*(x) = \max_{a \in A} Q^*(x, a) \tag{2.4}$$

$$Q^*(x, a) = r(x, a) + \gamma \sum_{x' \in X} P(x'|x, a) V^*(x'). \tag{2.5}$$

Ainsi nous pouvons, à partir de la fonction qualité optimale Q^*, définir une politique optimale déterministe dite *gloutonne* π^* telle que :

$$\pi^*(x) = \arg\max_{a \in A} Q^*(x,a), \forall x \in X.$$

La politique optimale π^* peut être définie de la même façon en utilisant V^*, r et P en utilisant (2.5) :

$$\pi^*(x) = \arg\max_{a \in A} \left\{ r(x,a) + \gamma \sum_{x' \in X} P(x'|x,a) V^*(x') \right\}, \forall x \in X. \qquad (2.6)$$

De l'équation (2.3) découle l'algorithme *value iteration* ou itération sur la valeur de [Bellman, 1957] — voir l'algorithme 2.1 — consistant à itérer le calcul de V pour tout $x \in X$ jusqu'à ce que la différence entre $V(x)$ et celui de l'itération précédente soit inférieure à une valeur $\epsilon \geq 0$ définie à l'avance et ce pour tout $x \in X$. La fonction valeur V ainsi calculée est une approximation de V^* à ϵ près.

Algorithme 2.1 Itération de la valeur

Initialiser $k \leftarrow 0$ et $V_k(x) \leftarrow 0, \forall x \in X$
repeat
  $\Delta \leftarrow 0$
  for all $x \in X$ **do**
    $V_{k+1}(x) \leftarrow \max_{a \in A} \left\{ r(x,a) + \gamma \sum_{x' \in X} P(x'|x,a) V_k(x') \right\}$
    $\Delta \leftarrow \max(\Delta, |V_k(x) - V_{k+1}(x)|)$
  end for
  $k \leftarrow k + 1$
until $\Delta < \epsilon$
return V_k

Pour illustrer un peu plus l'algorithme de l'itération de la valeur, reprenons l'exemple décrit par la figure 2.1 de la page 12 en supposant que le facteur de dépréciation est 0.99 et que $\epsilon = 0.01$. En premier lieu, la fonction valeur V est initialisée à 0 pour les treize états non-terminaux du MDP. Puis V va évoluer au cours des itérations successives pour converger vers V^* à ϵ

près :

1ère itération		2ème itération		3ème itération
$V((0,0)) = 0$		$V((0,0)) = 0$		$V((0,0)) = 4.165425$
$V((1,1)) = 0$		$V((1,1)) = 3.465$		$V((1,1)) = 3.465$
$V((1,2)) = 0$		$V((1,2)) = 3.465$		$V((1,2)) = 3.465$
$V((1,3)) = 0$		$V((1,3)) = 3.465$		$V((1,3)) = 3.465$
$V((1,4)) = 0$		$V((1,4)) = 3.96$		$V((1,4)) = 3.96$
$V((1,5)) = 0$		$V((1,5)) = 4.95$		$V((1,5)) = 4.95$
$V((1,6)) = 0$	$\Rightarrow$	$V((1,6)) = 5.94$	$\Rightarrow$	$V((1,6)) = 5.94$
$V((2,1)) = 1$		$V((2,1)) = 1$		$V((2,1)) = 1$
$V((2,2)) = 2$		$V((2,2)) = 2$		$V((2,2)) = 2$
$V((2,3)) = 3$		$V((2,3)) = 3$		$V((2,3)) = 3$
$V((2,4)) = 4$		$V((2,4)) = 4$		$V((2,4)) = 4$
$V((2,5)) = 5$		$V((2,5)) = 5$		$V((2,5)) = 5$
$V((2,6)) = 6$		$V((2,6)) = 6$		$V((2,6)) = 6$

La 4ème itération est identique à la 3ème et provoque donc l'arrêt de l'algorithme. La politique optimale générée par V^*, r et P est de relancer le dé si le score obtenu au premier jet est inférieur ou égal à 3 sinon nous ne faisons rien. Dans les autres états, la politique est contrainte par P.

Le même principe a été utilisé par Howard [1960] afin de créer l'algorithme *policy iteration* ou itération de la politique — voir l'algorithme 2.2 — se déroulant en deux étapes successives et itérées jusqu'à la convergence de la politique. La première étape se compose de l'équation de Bellman pour V^π :

$$V^\pi(x) = r(x, \pi(x)) + \gamma \sum_{x' \in X} P(x'|x, \pi(x)) V^\pi(x'), \forall x \in X$$

laquelle sera itérée jusqu'à convergence de la même manière que pour l'algorithme d'itération de la valeur, la politique π et la fonction valeur V^π initiales étant fixées arbitrairement. La seconde étape consiste à générer une nouvelle politique πi, à partir de la nouvelle fonction valeur, en utilisant l'équation (2.6). La première étape est ensuite renouvelée puis, la seconde et ce, jusqu'à ce que la nouvelle politique π obtenue ne diffère pas de celle obtenue à l'itération précédente. La première étape est communément appelée *policy evaluation* ou évaluation de la politique et la seconde étape, *policy improvement* ou amélioration de la politique.

Il est à remarquer que la condition d'arrêt de l'algorithme d'itération de la politique est beaucoup plus pratique que celle de l'algorithme d'itération de la valeur. En effet, il n'est pas rare qu'une convergence totale de V^* ne soit pas requise pour que la politique gloutonne associée soit optimale. Le choix de ϵ est donc crucial pour obtenir une politique optimale sans surcoût alors que, dans le cas l'algorithme d'itération de la politique, la condition d'arrêt est directement l'obtention d'une politique optimale. Par

Algorithme 2.2 Itération de la politique

Initialiser $\pi(x) \in A, \forall x \in X$
repeat
  $\pi' \leftarrow \pi$
  repeat
    $\Delta \leftarrow 0$
    $k \leftarrow 0$ et $V_k^\pi(x) \leftarrow 0, \forall x \in X$
    for all $x \in X$ **do**
      $V_{k+1}^\pi(x) \leftarrow r(x, \pi(x)) + \gamma \sum_{x' \in X} P(x'|x, \pi(x)) V_k^\pi(x')$
      $\Delta \leftarrow \max(\Delta, |V_{k+1}^\pi - V_k^\pi(x)|)$
      $k \leftarrow k + 1$
    end for
  until $\Delta < \epsilon$
  $\pi(x) \leftarrow \arg\max_{a \in A} \left\{ r(x, a) + \gamma \sum_{x' \in X} P(x'|x, a) V_k^\pi(x') \right\}, \forall x \in X$
until $\pi = \pi'$
return π

contre, chaque itération de l'algorithme d'itération de la politique est plus coûteux en calculs qu'une itération de l'algorithme d'itération de la valeur.

Dans l'exemple du jeu de dés — décrit en figure 2.1 de la page 12 — avec un facteur de dépréciation de 0.99 et $\epsilon = 0.01$, la politique optimale est générée en une itération puis nécessite une itération supplémentaire afin de confirmer son optimalité — c'est-à-dire que la politique générée par chaque itération est identique. Dans notre exemple, l'algorithme d'itération de la politique est moins efficace que l'algorithme d'itération de la valeur.

Dans ces deux algorithmes, une connaissance parfaite du modèle est requise que ce soit au niveau des probabilités de transition, que de la fonction récompense ce qui peut être une contrainte importante. De plus, il en ressort que chaque algorithme itère sur l'ensemble des états et des actions ce qui engendre un problème de passage à l'échelle pour des problèmes dont les états et/ou actions possèdent un grand nombre de dimensions. Bellman a nommé ce problème *curse of dimensionality* ou malédiction de la dimension, dans le cadre de la programmation dynamique du fait de la complexité exponentielle en le nombre de dimensions.

Néanmoins dans le pire des cas, ces deux méthodes de programmation dynamique peuvent trouver une politique optimale en temps polynomial en fonction du nombre d'états et d'actions. Ceci est déjà plus efficace que l'énumération des politiques stationnaires déterministes lors d'une recherche directe.

Lorsqu'un modèle n'est pas connu explicitement comme pour les algorithmes d'itération de la valeur ou d'itération de la politique, l'utilisation de quadruplets (x_t, a_t, r_t, x_{t+1}) issus directement du système — ou d'une simulation du système — est une alternative concrète pour apprendre une

politique optimale. Deux choix se présentent alors ; soit un modèle est inféré et ensuite utilisé pour apprendre une politique optimale, soit les échantillons sont utilisés directement pour apprendre une politique optimale. Nous allons maintenant nous attacher à la deuxième possibilité — la plus simple et la plus ancienne historiquement — au travers des algorithmes de type *Temporal-Difference* (TD) ou différence-temporelle et en premier lieu de TD(0).

TD(0) permet d'apprendre la fonction valeur V^π pour une politique π fixée à partir de quadruplets issus du système contrôlé par ladite politique. La mise à jour de la fonction valeur se fait ainsi ; soit un quadruplet (x_t, a_t, r_t, x_{t+1}) :

$$\delta = r_t + \gamma V^\pi(x_{t+1}) - V^\pi(x_t) \tag{2.7}$$

$$V^\pi_{t+1}(x_t) = V^\pi_t(x_t) + \alpha_t(x_t)\delta \tag{2.8}$$

avec $\alpha_t(x) \geq 0, \forall t \geq 0$ et $x \in X$. $\alpha_t(x)$ est le taux d'apprentissage permettant de contrôler l'amplitude des corrections effectuées pour un état donné au cours du temps. δ correspond à l'erreur de la différence temporelle et permet de réajuster la valeur courante de $V^\pi(x_t)$ proportionnellement au taux d'apprentissage $\alpha_t(x_t)$. Ainsi si $\delta > 0$ alors la valeur courante de $V^\pi(x_t)$ doit être revue à la hausse et, dans le cas contraire, — $\delta < 0$ — la valeur courante de $V^\pi(x_t)$ est trop optimiste et doit être revue à la baisse.

Nous pouvons observer que l'équation de mise à jour de $V^\pi(x)$ utilise sa valeur courante comme référence. Cette pratique est appelée *bootstrapping* dans le sens où les valeurs initiales ont une influence tout au long de l'exécution de l'algorithme. Nous verrons dans le chapitre suivant que les valeurs initiales peuvent influencer le temps de convergence en pratique.

De TD(0) et du principe de différence-temporelle découle un certain nombre d'algorithmes permettant l'apprentissage d'une politique optimale dont l'algorithme *Q-Learning* est l'un des plus anciens. Il a été mis au jour par [Watkins, 1989].

Comme nous l'avons vu précédemment TD(0) permet d'évaluer la valeur d'une politique. Il nous faut maintenant, un moyen de faire évoluer la politique vers une politique optimale, en nous servant de cette valeur. Q-Learning évalue une fonction qualité définissant une politique utilisée pour contrôler le système et tendra au cours du temps vers une politique optimale. L'algorithme Q-Learning est décrit par l'algorithme 2.3. Nous supposerons ici, que le système est épisodique.

Comme dans TD(0), δ correspond à l'erreur de différence-temporelle mais ici adapté à la fonction qualité. L'algorithme Q-Learning converge avec une probabilité de 1 vers une politique optimale, si les conditions suivantes sont respectées :

- Chaque couple état-action doit être visité une infinité de fois ;
- $\sum_{t\geq 0} \alpha_t(x) = +\infty, \forall x \in X$;
- $\sum_{t\geq 0} \alpha_t^2(x) < +\infty, \forall x \in X$.

Algorithme 2.3 Q-Learning

$Q(x,a) \leftarrow 0, \forall (x,a) \in (X,A)$
while ∞ **do**
  repeat
    $t \leftarrow 0$
    Initialiser x_0 comme état initial
    Choisir une action a_t
    Observer x_{t+1} et r_t
    $\delta = r_t + \gamma \max_{a' \in A} Q(x_{t+1}, a') - Q(x_t, a_t)$
    $Q(x_t, a_t) \leftarrow Q(x_t, a_t) + \alpha_t(x_t)\delta$
  until x_{t+1} n'est pas un état terminal
end while

Il est intéressant de constater que la méthode utilisée pour choisir l'action à appliquer au système n'entre pas en compte dans la convergence vers une politique optimale. Par contre, celle-ci influencera le temps de convergence vers la politique optimale. Le choix de la méthode de sélection de l'action fait émerger un problème commun à d'autres problèmes qui est le dilemme exploration/exploitation. Dans le cas de choix de l'action, l'exploration reviendrait à essayer des actions peu ou pas utilisées dans certains états pour, potentiellement, découvrir des séquences de récompenses avantageuses. Au contraire, l'exploitation serait d'appliquer une action déjà utilisée auparavant dans un certain état afin d'améliorer la précision de la fonction qualité pour ce couple état-action.

Intuitivement, nous aimerions explorer dans un premier temps puis exploiter dans un second temps. Il existe différentes méthodes de sélection dont les suivantes sont les plus connues :

La méthode gloutonne sélectionne l'action $a \in A$ qui maximise $Q(x,a)$. Aucune exploration n'est réalisée, seulement de l'exploitation.

La méthode ϵ-gloutonne sélectionne l'action $a \in A$ qui maximise $Q(x,a)$ avec une probabilité ϵ ou sélectionne au hasard une action $a \in A$ avec une probabilité $1-\epsilon$. Le paramètre ϵ sert donc, ici, à définir une proportion entre l'exploitation et l'exploration.

La méthode softmax sélectionne une action $a \in A$ avec une probabilité $P(a|x)$ tel que :

$$P(a|x) = \frac{e^{\frac{Q(x,a)}{\tau}}}{\sum_{a' \in A} e^{\frac{Q(x,a')}{\tau}}}.$$

La proportion entre l'exploitation et l'exploration est directement liée ici, à la qualité des actions régie par une distribution de Boltzmann et sa température τ. Ce paramètre τ, en diminuant doucement, permet de passer de l'exploration à l'exploitation. Nous basculons ainsi d'une

sélection uniforme à une sélection gloutonne.

D'autres méthodes plus complexes existent prenant en compte la précision de l'estimation de la fonction qualité dans l'espace des couples état-action pour favoriser l'exploration dans les zones les moins précises. Néanmoins le choix de la méthode utilisée est fortement dépendant du problème et, peut demander des connaissances en lien avec le système afin de sélectionner la méthode réduisant au maximum le temps de convergence.

Le dilemme exploration-exploitation peut-être vu plus simplement au travers du problème du *multi-armed bandit* — bandit à plusieurs bras. Dans ce problème, chaque bras du bandit possède une distribution propre mais inconnue. Ainsi, lorsqu'un bras est tiré, une récompense est tirée suivant la distribution qui lui est associée. Le but du problème étant de maximiser la somme des récompenses sur le long terme. Nous pouvons noter qu'il s'agit en fait d'un MDP avec un seul état ; chaque action étant un bras du bandit mais avec une fonction récompense stochastique et non déterministe comme décrite précédemment.

Nous voyons ainsi apparaître le dilemme exploration-exploitation naturellement. D'un côté, il faut explorer pour trouver le bras possédant l'espérance la plus élevée, même si cela demande d'essayer des bras qui, a priori, semblent sous-optimaux. De l'autre, il faut exploiter les bras optimaux afin d'affiner notre connaissance de leur espérance mais aussi, et surtout, pour maximiser la somme des récompenses à long terme. Ainsi, il est tout aussi contre-indiqué de faire de l'exploration ou de l'exploitation pure car, dans un cas nous tirerions un trop grand nombre de fois les bras sous-optimaux. Dans le cas contraire, nous pourrions nous focaliser sur un bras qui est sous-optimal. Il y a donc une répartition des tirages à effectuer de manière intelligente et ce, afin de maximiser la somme des récompenses à long terme.

Le dilemme exploration-exploitation est récurrent dans le domaine de l'apprentissage par renforcement surtout, si l'on cherche à obtenir des algorithmes efficaces dans l'utilisation de chaque échantillon. Dans le cas présent, les algorithmes comme le Q-Learning qui ne basent pas sur l'apprentissage d'un modèle — dits *model-free* ou sans modèle — soutirent moins d'informations de chaque échantillon que les algorithmes dits *model-based* ou basé sur un modèle comme Dyna [Sutton, 1991] ou RTDP [Barto et al., 1995]. Ce type d'algorithme maintient une estimation des probabilités de transition utilisée dans la mise à jour de la fonction valeur par exemple. Le dilemme exploration-exploitation revient ici entre explorer pour affiner l'estimation du modèle ou exploiter pour obtenir des récompenses plus élevées. Parmi ces algorithmes, nous pouvons citer R-Max [Brafman and Tennenholtz, 2001] ou encore UCRL2 [Jaksch et al., 2010] qui prennent en compte ce dilemme.

Un autre défi vient s'ajouter si le système envisagé est de grande dimension. Si le nombre d'états est grand voir infini, il peut être impossible de garder en mémoire l'estimation de la fonction valeur pour chaque état. Dans

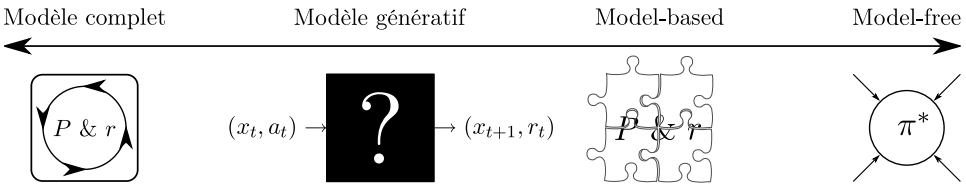


FIGURE 2.2 – Classement des modèles.

ce cas, des solutions ont été avancées, comme l'agrégation d'états ou encore l'utilisation de la régression pour approximer la fonction valeur en tous points de l'espace d'état. De plus, si l'espace d'état est continu, il peut être discrétisé pour obtenir un ensemble discret d'états, la méthode de discrétisation jouant alors un rôle prépondérant.

Nous avons vu que pour des algorithmes comme l'itération de la valeur, le modèle complet était requis. À l'inverse, un algorithme comme Q-Learning ne requiert aucune information sur le modèle mais ne cherche pas non plus à l'apprendre. Ainsi, nous pouvons observer que les algorithmes model-based sont un compromis entre les deux, en effet ils ne requièrent aucune information sur le modèle mais l'apprennent avec l'expérience acquise pour en faire usage. Cependant, il existe un intermédiaire entre avoir le modèle complet et apprendre le modèle ; il s'agit du modèle génératif. Celui-ci permet, étant donné un état et une action, d'obtenir un état successeur et la récompense associée. Nous pouvons voir le modèle génératif comme une boîte noire ; moins informatif qu'un modèle complet mais permettant d'obtenir des échantillons pour tout couple état-action. Ces différents modèles peuvent être classés suivant la force du modèle. Ce classement peut être observé sur la figure 2.2. La notion de modèle génératif est souvent associée à celle de la planification qui est abordée dans le chapitre suivant.

Chapitre 3

Planification et Optimisme

Ce chapitre aborde dans une première partie l'optimisme au travers d'exemples présents dans la littérature. Dans une deuxième partie, nous présenterons la planification dans le domaine de l'apprentissage par renforcement et, plus particulièrement, l'amélioration d'une politique existante ou l'obtention d'une politique par recherche arborescente. Enfin l'algorithme A* sera aussi présenté pour faire le lien avec l'algorithme de planification optimiste présenté dans le chapitre suivant.

3.1 L'optimisme dans l'apprentissage par renforcement

Heiner Müller disait :

> «L'optimisme est simplement un manque d'information.» [1]

Cette citation s'inscrit dans le domaine de l'apprentissage par renforcement au travers d'une démarche observée qui est d'avoir un comportement dit *optimiste*. Et cet optimisme se dévoile de différentes manières que nous allons vous présenter au moyen d'exemples issus de la littérature.

Si l'optimisme est un manque d'information, dans notre cas, nous allons l'utiliser pour favoriser la collecte d'informations ou à proprement parler l'exploration. L'optimisme est utilisé majoritairement comme une heuristique pour apporter une solution au dilemme exploration-exploitation [Szita and Lörincz, 2008]. Un principe a d'ailleurs été évoqué par [Lai and Robbins, 1985] en ces termes : «*Optimism in the Face of Uncertainty*» que l'on traduit en français par l'optimisme en cas d'incertitude. De ce principe découle une série d'algorithmes s'attaquant au problème du bandit à plusieurs bras dont l'algorithme UCB1 [Auer et al., 2002] — Upper Confidence Bound — est l'un des plus connus.

1. Traduit de l'allemand : «Optimismus ist nur ein Mangel an Information».

Comme expliqué dans le chapitre précédent, l'objectif dans le problème du bandit à plusieurs bras est de maximiser la somme des récompenses obtenues en tirant un bras à chaque pas de temps. Chaque bras distribue ses récompenses suivant une distribution de probabilités lui étant propre et nous étant inconnue. Un objectif intermédiaire est donc de trouver le bras ayant la meilleure espérance pour ainsi maximiser les gains à long terme. Plus formellement, un bandit à K-bras est défini par K variables aléatoires $(X_i)_{1\geq i\geq K}$. Chaque bras du bandit est identifié par un indice i et est indépendant des autres bras. Les récompenses d'un bras i sont les réalisations de X_i — $X_{i,0}, X_{i,1}, \ldots$ — et sont indépendantes et identiquement distribuées suivant une distribution inconnue d'espérance $\mu_i = \mathbb{E}X_i$ et de support $[0, r_{\max}]$. L'objectif est de maximiser $\sum_{t\geq 1} X_{I(t),t}$ avec $I(t)$ l'indice du bras tiré à l'instant t.

L'algorithme UCB1 définit une borne supérieur pour chaque bras i étant la somme de la moyenne des récompenses obtenues jusqu'à l'instant t et d'un intervalle de confiance supérieur sur l'estimation de cette moyenne. Cette intervalle de confiance est défini en fonction du nombre de fois que le bras i a été tiré et du nombre de tirages effectués t. Le bras tiré à chaque instant t est celui maximisant cette borne :

$$B_i = \hat{\mu}_{i,t} + r_{\max}\sqrt{\frac{2\ln t}{t_i}}$$

avec $\hat{\mu}_{i,t}$ la moyenne des récompenses obtenues jusqu'à l'instant t du bras i et t_i le nombre de tirages du bras i à l'instant t.

Le comportement optimiste est caractérisé ici par la définition d'une borne supérieure sur l'espérance de chaque bras et par la sélection du bras maximisant cette borne. L'intervalle de confiance composant la borne représente l'incertitude et le manque d'information sur le bras associé. Ainsi un bras peu tiré aura un intervalle de confiance large favorisant sa sélection. À l'inverse un bras tiré souvent verra son intervalle de confiance se réduire en amplitude, la précision de l'estimation de la moyenne associée grandissant ; c'est la partie exploration de la borne. De la même manière, un bras ayant une moyenne élevée mais un intervalle de confiance réduit aura toutes ses chances d'être tiré — la borne étant maximisée — et donc d'être exploité ; c'est la partie exploitation de la borne. La borne supérieure et sa maximisation induisent donc un équilibre entre l'exploration et l'exploitation.

L'optimisme est aussi présent sous une autre forme dans le domaine de l'apprentissage par renforcement. Nous avons parlé dans le chapitre précédent du bootstrapping, principe selon lequel la règle de mise à jour de la fonction valeur ou de la fonction qualité utilise sa valeur précédente. Ainsi la valeur initiale peut jouer un rôle dans certains cas. Dans la définition de l'algorithme 2.3 du Q-Learning à la page 18, la fonction qualité était initialisée à 0. Cette initialisation peut être faite de manière optimiste en donnant pour valeur $r_{\max}\frac{1}{1-\gamma}$ avec $r_{\max}$ la récompense maximum et γ le facteur de dépréciation

à tout état de la fonction valeur ou de la fonction qualité. Cette valeur est le maximum atteignable par la somme des récompenses dépréciées. Dans le cas du Q-Learning avec une sélection d'action du type Boltzmann, ces valeurs initiales incitent à l'exploration dans les zones non visitées. Ce bonus d'exploration — comme l'appelle Sutton dans Dyna [Sutton, 1991] — sera au fur et à mesure atténué par les mises à jour successives de la fonction qualité.

Une autre variante du bonus d'exploration est proposée par [Meuleau and Bourgine, 1999] où le bonus d'exploration est une mesure de l'estimation de la qualité d'un couple état-action et ce pour chaque couple état-action. Cette mesure est ajoutée à la récompense obtenue pour le couple état-action correspondant et est propagée de la même manière — en utilisant une méthode différence-temporelle par exemple. Ceci permet ainsi d'orienter l'exploration ainsi que l'exploitation vers les zones d'intérêts et cela de manière globale.

Dans le même principe, R-MAX [Brafman and Tennenholtz, 2001] utilise la récompense maximum pour initialiser les récompenses de l'approximation initiale du modèle d'où le nom de l'algorithme : R-MAX. De même, UCRL2 [Jaksch et al., 2010] sélectionne un MDP optimiste parmi un ensemble de MDP statistiquement probable. Cet ensemble est défini par l'ensemble des MDP dont les récompenses et les probabilités sont comprises dans des intervalles de confiance. Puis le MDP maximisant la moyenne de la somme des récompenses est sélectionné.

Nous pouvons donc voir l'optimisme dans le domaine de l'apprentissage par renforcement comme une méthode gloutonne mais sur des critères encourageant un équilibre entre exploration et exploitation dans le but de gérer si ce n'est résoudre le dilemme exploration-exploitation.

3.2 La planification dans les processus de décision markovien

La planification dans les processus de décision markovien consiste, à partir d'un état donné et d'un modèle génératif à faire une recherche avant — simuler les futurs possibles pour décider du présent — pour trouver l'action ou la séquence d'actions à appliquer au système réel modélisé par un MDP. Cette recherche de l'action optimale pour un état donné peut être vu comme une politique dont l'association état-action est générée à la demande par un calcul non-trivial. Si ce calcul utilise une donnée issue d'un calcul hors-ligne — comme une estimation de la fonction valeur —, nous parlerons alors d'amélioration de la politique existante — [Davies et al., 1998]. Dans le cas contraire, le calcul définira entièrement la politique. Il est à noter que la politique ainsi générée peut être stochastique.

Plus formellement, nous avons un système réel modélisé par un MDP (X, A, P, r) et se trouvant dans un état courant x_{crt}. Nous supposons avoir accès à un modèle génératif pour simuler le MDP et non un accès complet à celui-ci. Nous pourrons ainsi obtenir un état successeur $x_{t+1} \sim P(.|x_t, a_t)$ alors que les probabilités sous-jacentes resteront inconnues. Cependant les récompenses seront obtenues comme à l'accoutumée au travers de la fonction récompense $r : X \times A \to \mathbb{R}$ tel que $r_t = r(x_t, a_t)$. Un modèle génératif est plus puissant qu'un simple simulateur de trajectoire dans le sens où nous pouvons l'utiliser arbitrairement à partir de tout état contrairement au simulateur qui lui construit une séquence à partir d'un état initial jusqu'à un état terminal. Son utilisation est donc très utile dans la construction itérative d'un arbre des possibilités — *look-ahead tree* en anglais.

Le système réel évoluera d'un état à un autre par l'intermédiaire des actions provenant d'un algorithme de sélection d'action $\mathcal{A}$. Celui-ci retournera une action $a^{\mathcal{A}}$ étant donné un état initial x_0 correspondant à l'état courant x_{crt} du système réel. Cette action sera effectuée sur le système réel qui transitera dans un nouvel état $x_{\text{crt}+1} \sim P(.|x_{crt}, a^{\mathcal{A}})$. L'algorithme de sélection d'action sera de nouveau appelé avec un état initial correspondant au nouvel état courant $x_{\text{crt}+1}$ du système réel. Cette succession d'appels à l'algorithme de sélection d'action et de l'utilisation des actions retournées par celui-ci forme une séquence d'actions qui dans le meilleur des cas suivra une politique optimale. Le but d'un algorithme de sélection d'action sera donc, à partir d'un état initial x_0 et d'un modèle génératif de trouver l'action a^* maximisant l'espérance de la somme des récompenses dépréciées obtenues le long des séquences ayant x_0 pour premier état et a^* comme première action. L'algorithme Rollout se propose comme nous allons voir d'atteindre ce but.

3.2.1 L'algorithme Rollout

L'algorithme *Rollout* est dû à [Tesauro and Galperin, 1996] et permet de simuler des trajectoires de profondeur finie H suivant une politique initial π mais dont la première action est uniformément répartie entre celles appartenant à A. L'algorithme Rollout estime donc $\hat{Q}^{\pi}(x_0, a), \forall a \in A$ par une méthode dite de Monte-Carlo puis retourne l'action $a \in A$ maximisant $\hat{Q}^{\pi}(x_0, a)$. L'algorithme 3.1 décrit Rollout.

Algorithme 3.1 Rollout

for all $a \in A$ **do**
  $v \leftarrow 0$
  $x_1 \sim P(.|x_0, a)$
  $r_0 \leftarrow r(x_0, a)$
  for $n \leftarrow 1$ **to** N **do**
    for $t \leftarrow 1$ **to** H **do**
      $x_{t+1} \sim P(.|x_t, \pi(x_t))$
      $r_t \leftarrow r(x_t, \pi(x_t))$
      $v \leftarrow v + \gamma^t r_t$
    end for
  end for
  $\hat{Q}^{\pi}(x_0, a) \leftarrow r_0 + \frac{1}{N} v$
end for
return $\arg\max_{a \in A} \hat{Q}^{\pi}(x_0, a)$

Il est à observer que l'algorithme Rollout utilise une politique π préalablement calculée et sert donc à l'améliorer pour un état donné et ce en utilisant le temps disponible en ligne. L'algorithme Rollout peut aussi utiliser une estimation de la fonction valeur associée à la politique π pour chaque état x_H affinant ainsi l'estimation de $\hat{Q}^{\pi}$. La qualité de la politique initiale influençant la qualité de l'action retournée par l'algorithme Rollout, une bonne politique π est requise pour obtenir un contrôle s'approchant de l'optimal. La qualité de l'action est aussi affectée par les paramètres N et H de l'algorithme Rollout dont l'augmentation accroît la précision de l'estimation de la fonction qualité et donc de l'action optimale. [Tesauro and Galperin, 1996] suggère aussi d'ignorer les actions dont la valeur $\hat{Q}^{\pi}$ est en dehors d'un intervalle de confiance après un nombre fini de séquences simulées et ainsi de se concentrer sur les actions potentiellement optimales. Cette suggestion n'est pas sans rappeler le dilemme exploration-exploitation ainsi que le problème du bandit à plusieurs bras vus précédemment.

Bien que demandant une politique initiale à améliorer, l'algorithme Rollout a plusieurs avantages dont la faible occupation mémoire. En effet, au delà de la représentation en mémoire de la politique π, l'algorithme Rollout ne doit conserver en mémoire que les estimations de la fonction qualité pour

la politique π pour un état initial x_0 et pour toute action $a \in A$ ainsi que la trajectoire en cours de simulation. Donc si une représentation compacte de la politique π est disponible, l'algorithme Rollout peut traiter un espace d'état arbitrairement grand. De plus il peut être facilement parallélisé.

Si une politique initiale n'est pas connue, il est nécessaire de faire varier les actions le long des trajectoires simulées, ces variations générant alors un arbre des possibilités. Ces variations permettent d'explorer le sous-MDP issus de l'état initial. La question est alors de savoir comment sera exploré ce sous-MDP et donc la forme que devra prendre l'arbre émergeant de l'exploration. Cet arbre sera caractérisé par sa largeur et sa profondeur ainsi que par l'ordre dans lequel il aura été construit pour au final trouver une action proche de l'optimale pour un coût computationnel raisonnable. Un des premiers algorithmes abordant cette approche est l'algorithme Sparse Sampling.

3.2.2 L'algorithme Sparse Sampling

[Kearns et al., 2002] propose l'algorithme *Sparse Sampling* construisant un arbre uniforme dont la profondeur H et la largeur N sont calculées en fonction de la précision voulue ϵ, du facteur de dépréciation γ, de la récompense maximum $r_{\max}$ et du facteur de branchement de l'arbre, K — typiquement $K = |A|$. N et H sont définis ainsi :

$$H = \left\lceil \log_\gamma(\frac{\lambda}{V_{\max}}) \right\rceil, \tag{3.1}$$

$$N = \frac{V_{\max}^2}{\lambda^2}\left(2H \log \frac{KHV_{\max}^2}{\lambda^2} + \log \frac{r_{\max}}{\lambda}\right) \tag{3.2}$$

avec $\lambda = \epsilon(1-\gamma)^2/4$ et $V_{\max} = r_{\max}\frac{1}{1-\gamma}$. ϵ borne la différence entre l'estimation de la fonction valeur par l'algorithme Sparse Sampling et la fonction valeur optimale dans l'état initial x_0 tel que :

$$|V^*(x_0) - V^{\mathcal{A}}(x_0)| \leq \epsilon.$$

L'algorithme Sparse Sampling construit un arbre uniforme — dans le sens où toutes les feuilles de l'arbre sont à la même profondeur — dont le nœud racine est associé à l'état initial x_0. Les nœuds enfants du nœud racine sont générés par l'appel au modèle génératif et ce, N fois pour chacune des actions $a \in A$. Les enfants de ces nœuds sont générés de la même façon jusqu'à ce que leurs profondeurs soient égales à H et donc que le nombre de transitions pour toutes les séquences démarrant de x_0 soit égal à H. Pour faciliter l'implémentation et comme H est connu à l'avance, l'algorithme Sparse Sampling construit l'arbre uniforme de manière récursive comme décrit par l'algorithme 3.2.

La figure 3.1 illustre un arbre partiel généré avec l'algorithme Sparse Sampling avec $A = \{a_1, a_2\}$. Nous pouvons ainsi observer que chaque couple

Algorithme 3.2 Sparse Sampling

function `buildTree`(x_h, h)
  if $h = H$ **then**
    return 0
  end if
  for all $a \in A$ **do**
    $\hat{Q}_h(x_h, a) \leftarrow 0$
    for $n \leftarrow 1$ **to** N **do**
      $x_{h+1} \sim P(.|x_h, a)$
      $r_h \leftarrow r(x_h, a)$
      $\hat{Q}_h(x_h, a) \leftarrow \hat{Q}_h(x_h, a) +$ `buildTree`$(x_{h+1}, h+1)$
    end for
    $\hat{Q}_h(x_h, a) \leftarrow r_h + \frac{1}{N}\gamma\hat{Q}_h(x_h, a)$
  end for
  return $\max_{a \in A}\hat{Q}_h(x_h, a)$

function `sparseSampling`(x_0)
  Calculer H et N suivant les équations (3.1) et (3.2) respectivement
  `buildTree`$(x_0, 0)$
  return $\arg\max_{a \in A}\hat{Q}_0(x_0, a)$

état-action est bien échantillonné N fois au travers du modèle génératif et ce, pour créer des séquences de H transitions.

Bien que la complexité de l'algorithme Sparse Sampling ne dépend pas de $|X|$, elle dépend de ϵ et de γ :

$$\left(\frac{K}{\epsilon(1-\gamma)}\right)^{\left(\frac{1}{1-\gamma}\log\left(\frac{1}{\epsilon(1-\gamma)}\right)\right)}.$$

Ainsi plus la précision requise augmente — ϵ proche de 0 — ou plus les futures récompenses sont importantes pour la prise de décision immédiate — γ proche de 1 — et plus le nombre d'appels au modèle génératif augmente rendant l'algorithme Sparse Sampling impraticable. [Kearns et al., 2002] suggère cependant de faire diminuer N en fonction de h. En effet les récompenses étant dépréciées, celles les plus éloignées de l'état initial influenceront moins sur l'estimation des $\hat{Q}_0(x_0, a), \forall a \in A$ que celles qui lui sont proches. Ainsi l'une des solutions est de définir N_h tel que :

$$N_h = \gamma^{2h}N$$

et permet ainsi de faire diminuer le nombre d'appels au modèle génératif sans faire baisser significativement la précision de l'estimation. Cependant nous pourrions être tenté de choisir arbitrairement H et N mais [Péret and

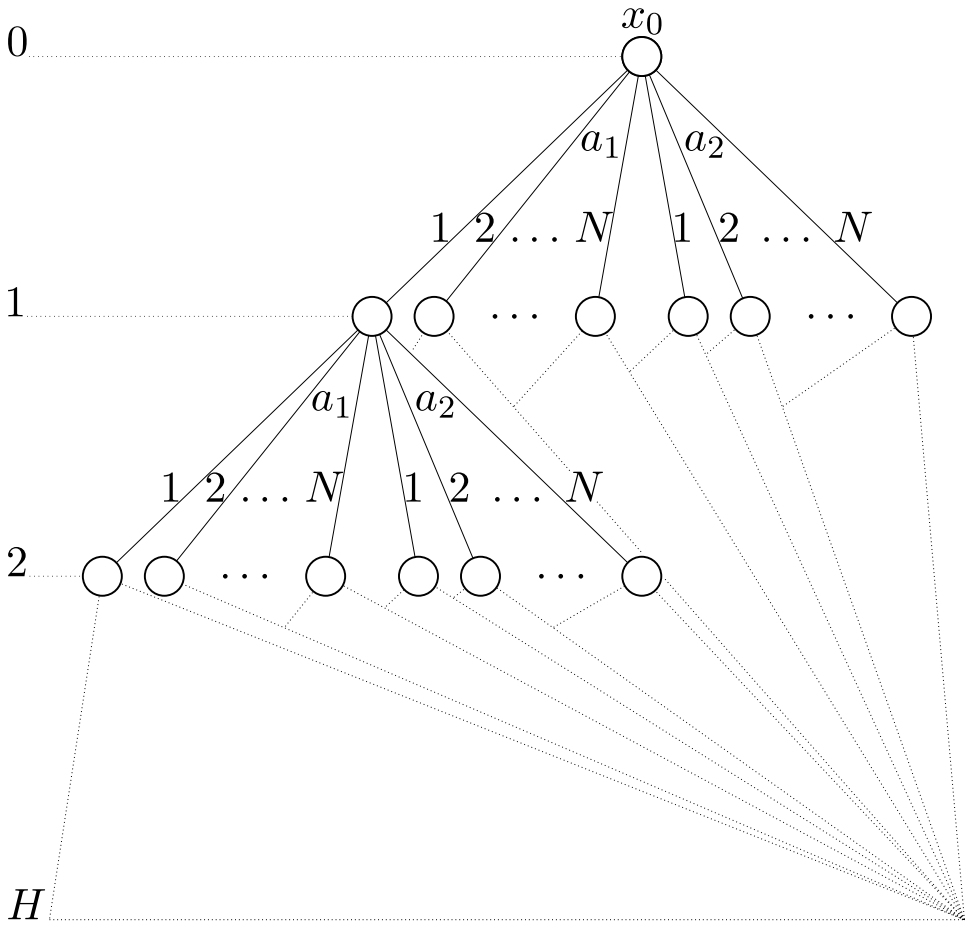


FIGURE 3.1 – Exemple d'un arbre partiel généré avec l'algorithme Sparse Sampling. Les triangles en pointillés sont les parties non-développées de l'arbre uniforme.

Garcia, 2004] a montré qu'augmenter H avec un N fixé augmente l'erreur sur l'estimation de la fonction qualité en l'état initial — problème nommé *look-ahead pathologies*. De plus, l'algorithme Sparse Sampling n'est pas adapté dans le cas où l'action doit être retournée dans un temps imparti ou tout autre contrainte qui n'est pas connue à l'avance. En effet le calcul de H et N se faisant en amont de la construction de l'arbre des possibilités, ces valeurs ne peuvent pas être calculées en prenant en compte ces contraintes mais uniquement en fonction de la précision requise et du facteur de dépréciation qui lui est partie intégrante du problème ; Sparse Sampling n'est pas *anytime* par nature.

Un algorithme est anytime si il est capable de retourner une solution à tout instant dont la qualité n'est peut-être pas optimale mais qui, cependant, augmentera au fur et à mesure que des ressources computationnelles supplé-

mentaires lui seront allouées. Dans le cadre de la planification, cette propriété est intéressante car si le système réel est contrôlé en temps réel alors le temps alloué à l'algorithme de sélection d'action est dicté par le système réel et est donc un paramètre potentiellement inconnu à l'avance. Une solution pour rendre l'algorithme Sparse Sampling anytime serait de d'augmenter progressivement H — *Iterative-Deepening* — comme décrit dans [Korf, 1985] mais indirectement, en jouant sur la précision requise ϵ. Cependant l'algorithme Sparse Sampling étant déjà gourmand en ressources computationnelles, cette approche n'est pas viable.

L'algorithme Sparse Sampling construit un arbre des possibilités uniforme possédant une profondeur finie définie à l'avance. Cependant toutes les actions admissibles dans un même état n'ont pas la même importance au travers de la fonction qualité associée et donc ne devraient pas être échantillonnées uniformément mais en fonction de l'estimation de la fonction qualité au cours de l'exécution de l'algorithme. Ceci n'est pas sans rappeler la suggestion de [Tesauro and Galperin, 1996] à propos de l'algorithme Rollout et de l'allocation adaptative des simulations par action en fonction de leur qualité courante. Nous allons voir maintenant comment [Chang et al., 2005] a associé l'algorithme Sparse Sampling avec l'algorithme UCB1 pour créer l'algorithme Adaptive Multistage Sampling permettant l'échantillonnage des actions dans un même état dans l'arbre des possibilités généré.

3.2.3 L'algorithme Adaptive Multistage Sampling

L'algorithme *Adaptive Multistage Sampling* (AMS) de [Chang et al., 2005] construit un arbre des possibilités dont la profondeur H est définie à l'avance dans le but de trouver une action optimale dans un MDP à horizon fini H. De la même manière que les deux algorithmes précédents, l'utilisation de l'algorithme AMS permet de sélectionner l'action à utiliser dans l'état courant du système réel. Cependant contrairement à [Kearns et al., 2002], H est ici considéré comme un paramètre du problème et permet à l'algorithme AMS de définir une politique stochastique pour un problème de contrôle à horizon fini H.

L'arbre uniforme généré par l'algorithme AMS diffère de celui généré par l'algorithme Sparse Sampling au niveau de l'échantillonnage des actions. En effet dans l'algorithme Sparse Sampling, chaque action $a \in A$ d'un état $x_h, H > h \geq 0$ appartenant à l'arbre généré est échantillonnée N fois. Dans l'algorithme AMS ce nombre d'échantillons — $N|A|$ si nous voulons être consistant avec l'algorithme Sparse Sampling — sera réparti entre les différentes actions dans un état $x_h, H > h \geq 0$ grâce à une méthode dérivée de l'algorithme UCB1. Cette pratique aura pour effet de concentrer les échantillons sur les actions les plus intéressantes vis à vis de l'estimation de la fonction qualité pour ce couple état-action tout en explorant les actions dont la précision de l'estimation est plus faible. L'algorithme UCB1 servira donc à

équilibrer l'exploration et l'exploitation dans la répartition de l'effort computationnel affecté à chaque état appartenant à l'arbre des possibilités généré.

L'algorithme AMS est décrit par l'algorithme 3.3 où $S_{x,a}$ est un multi-ensemble contenant les états successeurs générés par le modèle génératif par l'application de l'action a dans l'état x et $n_{x,a}$ est le nombre d'appels au modèle génératif par le couple état-action (x, a).

Algorithme 3.3 Adaptive Multistage Sampling

function `buildTree`(x_h, h)
  if $h = H$ **then**
    return 0
  end if
  for all $a \in A$ **do**
    $x_{h+1} \sim P(.|x_h, a)$
    $r_h \leftarrow r(x_h, a)$
    $\hat{V}_{h+1}(x_{h+1}) \leftarrow$ `buildTree`$(x_{h+1}, h+1)$
    $S_{x_h,a} \leftarrow x_{h+1}$
    $n_{x_h,a} \leftarrow 1$
    $\hat{Q}_h(x_h, a) \leftarrow r_h + \gamma \hat{V}_{h+1}(x_{h+1})$
  end for
  while $\sum_{a \in A} n_{x_h,a} \neq N$ **do**
    $a \leftarrow \arg\max_{a \in A} \hat{Q}_h(x_h, a) + \sqrt{\frac{2 \ln \sum_{a \in A} n_{x_h,a}}{n_{x_h,a}}}$
    $x_{h+1} \sim P(x_h, a)$
    $r_h \leftarrow r(x_h, a)$
    $\hat{V}_{h+1}(x_{h+1}) \leftarrow$ `buildTree`$(x_{h+1}, h+1)$
    $S_{x_t,a} \leftarrow S_{x_t,a} \cup \{x_{h+1}\}$
    $n_{x_h,a} \leftarrow n_{x_h,a} + 1$
    $\hat{Q}_h(x_h, a) \leftarrow r_h + \gamma \frac{1}{n_{x_h,a}} \sum_{x' \in S_{x_t,a}} \hat{V}_{h+1}(x')$
  end while
  $\hat{V}_h(x_h) \leftarrow \sum_{a \in A} \frac{n_{x_h,a}}{N} \hat{Q}_h(x_h, a)$
  return $\hat{V}_h(x_h)$

function `AMS`(x_0)
  `buildTree`$(x_0, 0)$
  return $\arg\max_{a \in A} \hat{Q}_0(x_0, a)$

La figure 3.2 à la page 31 illustre un arbre partiel généré par l'algorithme AMS avec $A = \{a_1, a_2\}$. Nous remarquerons que contrairement à l'algorithme Sparse Sampling, chaque couple état-action x, a a un nombre d'échantillons $n_{x,a}$ qui lui est propre à cause de la répartition des N échantillons pour chaque état entre les $|A|$ actions par la méthode dérivée de l'algorithme

UCB1.

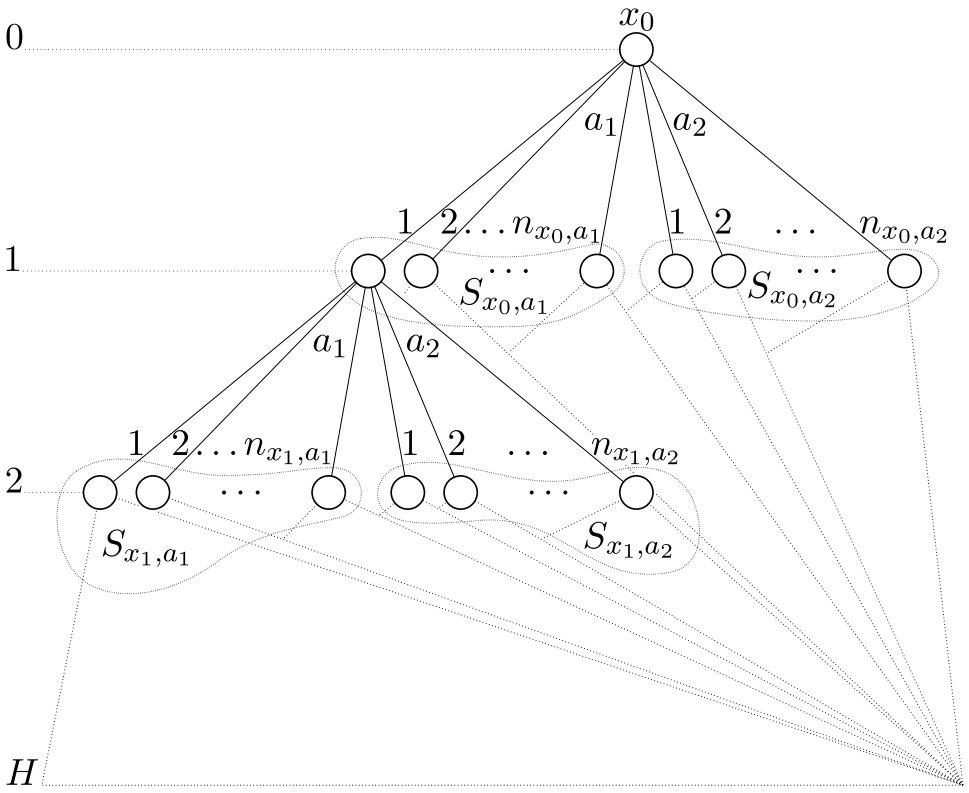


FIGURE 3.2 – Exemple d'un arbre partiel généré avec l'algorithme AMS. Les triangles en pointillés sont les parties non-développées de l'arbre uniforme.

L'algorithme AMS permet d'avoir une répartition non-uniforme des ressources computationnelles sur les actions pour chaque état mais comme l'algorithme Sparse Sampling, il n'est pas par nature anytime — N doit être fixé à l'avance. L'algorithme UCT que nous allons voir maintenant construit itérativement un arbre des possibilités asymétrique sans contraintes fixées à l'avance sur sa largeur ou sa profondeur ni même sur l'ordre dans lequel les nœuds seront développés.

3.2.4 L'algorithme UCB applied to Tree

L'algorithme *UCB applied to Tree* (UCT) de [Kocsis and Szepesvári, 2006] produit une succession de séquences d'actions choisies par une méthode dérivée de l'algorithme UCB1. Une séquence s'arrête lorsqu'un état non-rencontré précédemment est atteint. Cette succession de séquences d'actions avec regroupement des états identiques forme ainsi au fur et à mesure un arbre des possibilités dont la largeur et la hauteur augmentent au cours des itérations. Ce processus de génération est continué jusqu'à ce qu'une

condition d'arrêt soit atteinte — budget computationnel atteint, mémoire disponible épuisée, temps écoulé, etc... L'action maximisant l'estimation de la fonction qualité pour l'état initial — étant l'état courant du système réel — est alors retournée.

L'algorithme UCT est la combinaison entre l'algorithme Rollout et l'algorithme UCB1. En effet, la politique initiale utilisée dans l'algorithme Rollout pour diriger les séquences est ici remplacée par une politique basée sur l'algorithme UCB1 permettant d'explorer les actions peu utilisées mais aussi d'exploiter les actions prometteuses afin d'affiner l'estimation de la fonction qualité associée. L'algorithme 3.4 à la page 32 décrit l'algorithme UCT où $n_{x,a}$ est le nombre d'appels courant au modèle génératif pour le couple état-action (x, a), n_x est le nombre d'appels courant au modèle génératif pour l'état x et $C_p > 0$ est une constante.

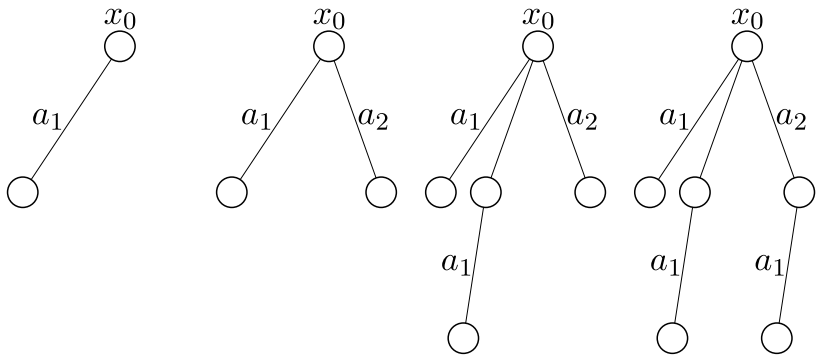
Algorithme 3.4 UCB applied to Tree

function `buildSequence`(x_h, h)
  if $\exists a \in A, n_{x_h,a} = 0$ **then**
    $r_h \leftarrow r(x_h, a)$
    $\hat{Q}(x_h, a) = r_h$
    $n_{x_h,a} \leftarrow 1$
    $n_{x_h} \leftarrow n_{x_h} + 1$
    return $\max_{a \in \{a \in A | n_{x_h,a} \neq 0\}} \hat{Q}(x_h, a)$
  end if
  $a \leftarrow \arg\max_{a \in A} \hat{Q}(x_h, a) + 2C_p \sqrt{\frac{\ln n_{x_h}}{n_{x_h,a}}}$
  $x_{h+1} \sim P(.|x_h, a)$
  $r_h \leftarrow r(x_h, a)$
  $q_{x_h,a} \leftarrow q_{x_h,a}$+ `buildSequence`$(x_{h+1}, h+1)$
  $\hat{Q}(x_h, a) \leftarrow r_h + \frac{1}{n_{x_h,a}} \gamma q(x_h, a)$
  $n_{x_h,a} \leftarrow n_{x_h,a} + 1$
  $n_{x_h} \leftarrow n_{x_h} + 1$
  return $\max_{a \in A} \hat{Q}(x_h, a)$

function `UCT`(x_0)
  repeat
    `buildSequence`$(x_0, 0)$
  until Une condition d'arrêt est vraie
  return $\arg\max_{a \in A} \hat{Q}(x_0, a)$

La figure 3.3 à la page 33 illustre un arbre asymétrique généré par l'algorithme UCT au travers de quatre séquences/itérations. Les sous-figures 3.3a et 3.3b montrent l'état initial x_0 et les deux états successeurs obtenus par l'utilisation des 2 actions a_1 et a_2. En effet, l'état initial étant une feuille

de l'arbre, avant que la sélection de l'action se fasse par la méthode dérivée de l'algorithme UCB1, il faut que toutes les actions soient sélectionnées une première fois. Ensuite la sous-figure 3.3c nous montre la troisième séquence dont la première action est a_1 et crée un nouvel état qui, étant une feuille, choisi arbitrairement l'action a_1 pour créer de nouveau un état. Enfin la quatrième séquence — illustrée par la sous-figure 3.3d — a pour première action a_2 dont l'utilisation au travers du modèle génératif retourne le même état que précédemment. Cet état étant une feuille, l'action a_1 est choisie arbitrairement et son utilisation engendre un nouvel état.



(a) La 1reséquence. (b) La 2^{e}séquence. (c) La 3^{e}séquence. (d) La 4^{e}séquence.

FIGURE 3.3 – Exemple d'exécution de l'algorithme UCT.

L'algorithme UCT et l'algorithme AMS ont comme point commun d'utiliser tous les deux l'algorithme UCB1, cependant certaines différences sont à noter. Dans un premier temps, l'algorithme AMS procède à une recherche en profondeur — *deep-first search* en anglais — explorant ainsi les feuilles en premier puis remontant vers la racine contrairement à l'algorithme UCT qui, étant basé sur l'algorithme Rollout simule des trajectoires de la racine aux feuilles. Dans un deuxième temps, l'algorithme AMS alloue, comme nous avons vu précédemment, N itérations à l'algorithme UCB1 en chacun des états rencontrés. Si N est suffisamment grand, l'instance de l'algorithme UCB peut converger vers la vraie action optimale permettant alors de faire remonter à la profondeur précédente une valeur théoriquement plus proche de l'optimale que celle provenant d'une instance de l'algorithme UCB1 qui n'a pas le temps de converger comme c'est le cas pour l'algorithme UCT. En effet, l'algorithme UCT fait remonter les valeurs obtenues le long des trajectoires alors même que les instances de l'algorithme UCB1 n'ont pas eu encore le temps nécessaire pour converger. Ainsi, contrairement à l'algorithme AMS possédant une garantie théorique sur ses performances, l'algorithme UCT peut ne pas converger en temps fini. De plus, [Coquelin and Munos, 2007] montre que dans certains cas le regret de l'algorithme UCT est moins bon que celui obtenu par une exploration uniforme. Ainsi bien que l'algorithme

UCT ait un comportement imprévisible, il a permis d'obtenir de très bons résultats sur certaines applications comme le Computer Go [Gelly and Wang, 2006].

Nous pouvons aussi remarquer que l'algorithme UCT est particulièrement efficace lorsque le nombre d'états successeurs pour tout état est faible. C'est à dire lorsque :

$$\left|\left\{x' \in X | P(x'|x,a) \neq 0, \forall a \in A\right\}\right| \ll |X|, \forall x \in X.$$

En effet, si le nombre d'états successeurs est faible alors la probabilité au cours du temps de retomber sur un état déjà visité, et donc appartenant à l'arbre, augmente permettant alors de sélectionner grâce à la méthode dérivée de l'algorithme UCB1 l'action à appliquer. Cette propriété est reprise comme hypothèse par [Buşoniu et al., 2011] pour définir l'algorithme Optimistic Planning for Sparsely Stochastic Systems que nous allons décrire maintenant.

3.2.5 L'algorithme Optimistic Planning for Sparsely Stochastic Systems

L'algorithme *Optimistic Planning for Sparsely Stochastic Systems* (OPSS) proposé par [Buşoniu et al., 2011] exploite le nombre restreint de successeurs pour construire un arbre des possibilités en utilisant une approche optimiste inspirée en partie de l'algorithme Optimistic Planning (OP) de [Hren and Munos, 2008] présenté dans le chapitre suivant. Il est à noter cependant que l'algorithme OPSS demande un accès au modèle complet et en particulier aux probabilités de transition.

L'algorithme OPSS construit un arbre des possibilités itérativement en sélectionnant successivement une feuille de l'arbre pour l'étendre. Étendre une feuille revenant ici à attacher en tant qu'enfant tous les états successeurs à l'état associé à cette feuille et ce, pour chacune des actions. De fait, si le nombre maximum d'états successeurs pour une même action est N, étendre une feuille créera au plus $N|A|$ nouvelles feuilles. Le point centrale de l'algorithme est donc de savoir quelle feuille doit être étendue.

Pour trouver la feuille à étendre, un sous-arbre est d'abord extrait de l'arbre généré par les précédentes itérations en sélectionnant récursivement, à partir du nœud racine, les nœuds associés aux états x_{t+1} enfants du nœud associé x_t et issus de l'action retournée par $B(x_t)$:

$$\begin{array}{rcll} b(x_t) & = & \begin{cases} R(x_t) + \frac{\gamma^t}{1-\gamma} & \text{Si } x_t \text{ est associé à} \\ & & \text{une feuille} \\ \max_{a \in A} \sum_{x_{t+1}} P(x_{t+1}|x_t,a) b(x_{t+1}) & \text{Sinon} \end{cases} \\ B(x_t) & = & \arg\max_{a \in A} \sum_{x_{t+1}} P(x_{t+1}|x_t,a) b(x_{t+1}) & \end{array}$$

avec $R(x_t)$ la somme des récompenses dépréciées le long de la trajectoire allant de x_0 à x_t.

Une fois ce sous-arbre obtenu, la feuille appartenant à ce sous-arbre maximisant $P(x_t)\frac{\gamma^t}{1-\gamma}$ avec x_t l'état associé à la feuille et $P(x_t)$ la probabilité d'atteindre x_t depuis x_0 est étendue. L'algorithme OPSS est décrit par l'algorithme 3.5.

Algorithme 3.5 Optimistic Planning for Sparsely Stochastic Systems

Initialiser l'arbre des possibilités contenant uniquement le nœud racine associé à x_0
repeat
  Obtenir un sous-arbre grâce à l'utilisation de la borne b
  Sélectionner la feuille associée à x_t de ce sous-arbre qui maximise $P(x_t)\frac{\gamma^t}{1-\gamma}$
  Étendre la feuille sélectionnée pour agrandir l'arbre des possibilités
until Une condition d'arrêt est vraie
return L'action a qui maximise $Q(x_0, a)$ définie sur les états appartenant à l'arbre généré

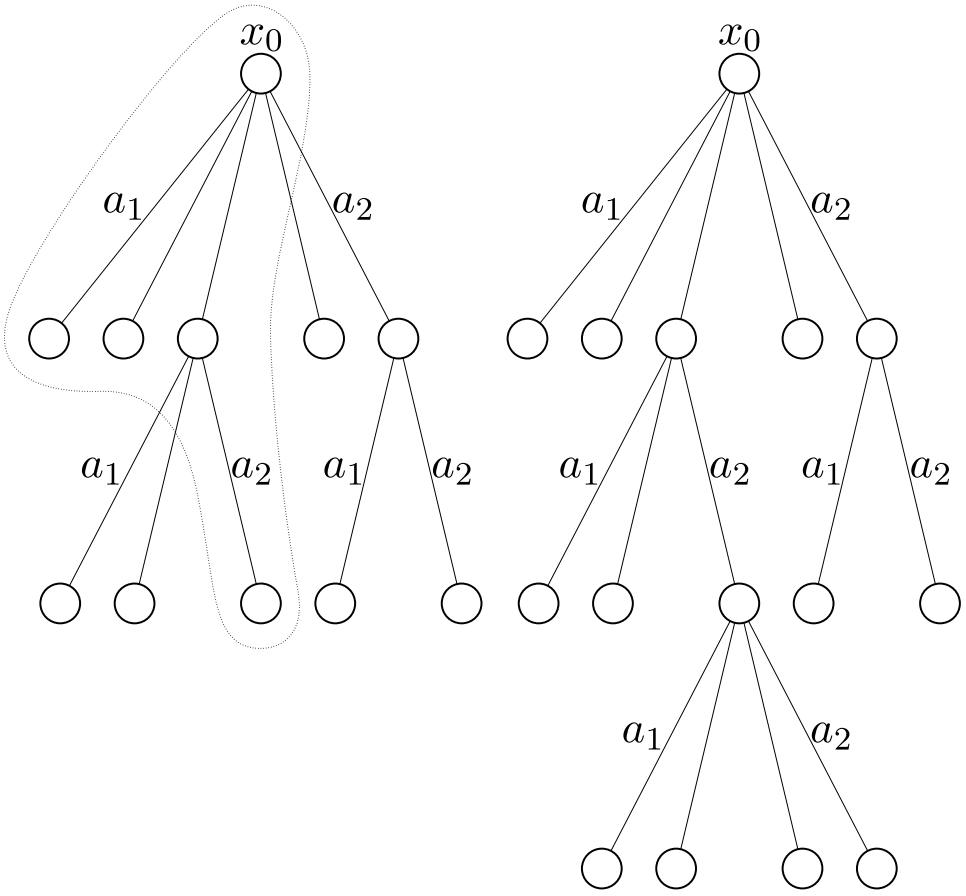
La figure 3.4 illustre la sélection d'un sous-arbre en 3.4a et la sélection d'une feuille dans ce sous-arbre ainsi que l'ajout des états successeurs à l'état associé à la feuille sélectionnée en 3.4b.

L'algorithme OPSS n'est pas sans rappeler l'algorithme UCT dans sa construction itérative de l'arbre des possibilités à ce détail près que l'algorithme OPSS génère tous les états successeurs d'un état associé à une feuille lorsque celle-ci est étendue grâce au modèle complet. Ce détail fait que le comportement anytime de l'algorithme OPSS est un peu moins fin que celui de l'algorithme UCT. En effet, là où l'algorithme UCT ajoute une feuille par itération, l'algorithme OPSS ajoute au minimum $|A|$ feuilles par itération.

De plus, l'algorithme OPSS, et contrairement à l'algorithme UCT, n'opère aucun test de pré-existence d'un état successeur ce qui dans le cas d'un espace d'état avec un nombre de dimensions élevé peut être avoir un coût non-négligeable. À l'opposé se trouve l'algorithme Open-Loop Optimistic Planning de [Bubeck and Munos, 2010] qui ne prend en considération que les séquences d'actions sans regard aucun pour les séquences d'états sous-jacents et que nous allons maintenant décrire.

3.2.6 L'algorithme Open-Loop Optimistic Planning

L'algorithme *Open-Loop Optimistic Planning* (OLOP) de [Bubeck and Munos, 2010] est un algorithme de planification en boucle ouverte dans le sens où les séquences d'états obtenues lors du processus de planification ne sont pas prises en compte dans le processus décisionnel. En effet, [Bubeck and Munos, 2010] n'utilise que les séquences d'actions comme point de repère pour évaluer la fonction qualité pour l'état initial. Cependant une planification optimale en boucle ouverte n'est pas optimale en boucle fermée.



(a) Extraction du sous-arbre à l'aide de B. (b) Sélection de la feuille qui est ensuite étendue.

FIGURE 3.4 – Exemple de la sélection d'une feuille à étendre.

L'algorithme OLOP évalue ainsi toutes les séquences d'actions de longueur finie $L = \lceil \log M/(2 \log 1/\gamma) \rceil$ avec M le plus grand entier tel que $M\lceil \log M/(2 \log 1/\gamma) \rceil \leq n$ avec M le nombre d'itération de l'algorithme et n le nombre maximum d'appels au modèle génératif qui sera donné en entrée. Nous pouvons ainsi noter que l'algorithme OLOP n'est pas anytime.

L'arbre généré par l'algorithme OLOP contient des nœuds qui ne sont pas associés avec des états mais avec des actions. Ainsi chaque nœud de l'arbre servira à garder en mémoire les statistiques de l'action associée au sein d'une séquence d'actions précise. L'algorithme OLOP définit une borne sur chacune des séquences d'actions finies possibles et choisit à chaque itération de simuler la séquence d'actions ayant la borne la plus élevée. Cette borne est définie récursivement sur l'arbre généré de façon à mutualiser les simulations effectuées pour deux séquences d'actions différentes mais ayant en commun

leurs premières actions. La borne sur une séquence $a \in A^L$ est définie pour tout $m \in \{1, \ldots, M\}$ et $h \in \{1, \ldots, L\}$ tel que :

$$B_a(m) = \inf_{h \in \{1,\ldots,L\}} U_{a_{1:h}}(m)$$

avec $a_{1,h}$ les h premières actions de la séquence a et $U_a(m)$ pour $a \in A^h$ la borne supérieure de confiance sur la séquence d'action a tel que :

$$U_a(m) = \begin{cases} +\infty & \text{si } T_a(m) = 0, \\ \sum_{t=1}^{m} \left(\gamma^t \hat{\mu}_{a_{1:t}}(m) + \gamma^t \sqrt{\frac{2 \log M}{T_{a_{1:t}}(m)}} \right) + \frac{\gamma^{h+1}}{1-\gamma} & \text{sinon.} \end{cases}$$

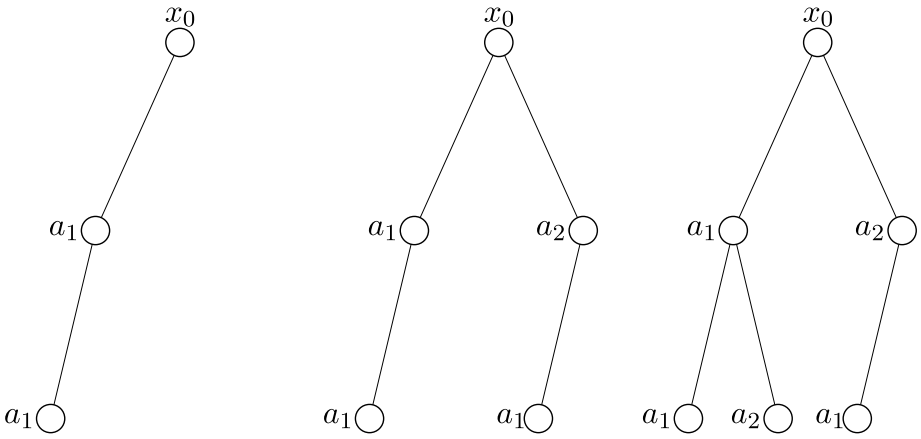
$\hat{\mu}_a(m)$ avec $a \in A^h$ est la moyenne empirique des récompenses obtenues pour a^h appartenant à toutes les séquences dont $a_{1:h} = a$ parmi les m itérations précédentes. $T_a(m)$ avec $a \in A^h$ est le nombre de fois que la séquence a a été simulée pendant les m précédentes itérations. L'action retournée par l'algorithme OLOP est la première action ayant été le plus simulée. L'algorithme OLOP est décrit par l'algorithme 3.6.

Algorithme 3.6 Open-Loop Optimistic Planning

Calculer M et L en fonction de n
$m \leftarrow 0$
Initialiser $B_a(m) = +\infty, \forall a \in A^L$
repeat
  $a = \arg\max_{a \in A^L} B_a(m)$
  Simuler la séquence d'actions a
  $m \leftarrow m + 1$
  Mettre à jour les $B_a(m), \forall a \in A^L$
until $m = M$
return $\arg\max_{a \in A} T_a(m)$

La figure 3.5 illustre l'exécution sur quatre itérations de l'algorithme OLOP pour $L = 2$ et $A = \{a_1, a_2\}$. À la première itération, la séquence a_1a_1 est sélectionné arbitrairement puisque toutes les bornes $B_a(0), \forall a \in A^2$ sont égales à $+\infty$. La séquence est simulée et les bornes pour les séquences a_1a_1 et a_1a_2 sont remises à jour. En effet, la valeur $U_{a_1}(1)$ est commune au deux séquences donc la borne pour la séquence a_1a_2 peut être aussi mise à jour. À la deuxième itération, la séquence a_2a_1 est choisie car sa borne est encore égale à $+\infty$. Elle est simulée et, comme précédemment, la borne des séquences a_2a_1 et a_2a_2 est remise à jour. Enfin à la troisième itération si $\hat{\mu}_{a_1}(2) \geq \hat{\mu}_{a_2}$ alors la séquence a_1a_2 est choisie arbitrairement entre a_1a_1 et a_1a_2 puisque $B_{a_1a_1}(2) = B_{a_1a_2}(2)$ principalement à cause de $T_{a_1}(2) = 1$ de sorte que $U_{a_1}(2) \leq U_{a_1a_1}(2) < U_{a_1a_2}(2) = +\infty$ et donc $B_{a_1a_1}(2) = B_{a_1a_2}(2) = U_{a_1}(2)$. Nous voyons ainsi apparaître une alternance entre la minimisation des U_a à partir de la séquence venant d'être simulée pour le

calcul des bornes et la maximisation de cette borne pour définir la séquence prochainement simulée.



(a) 1re itération. (b) 2e itération. (c) 3e itération.

FIGURE 3.5 – Exemple d'exécution de l'algorithme OLOP.

Le calcul à l'avance de la longueur et du nombre d'itérations n'est pas sans rappeler l'algorithme Sparse Sampling bien que le comportement de sélection d'une séquence d'actions en vu de la simuler une nouvelle fois pour affiner la borne et obtenir une meilleur estimation de la qualité de la première action peut être mis en parallèle avec celui de l'algorithme UCT. Les principales caractéristiques de l'algorithme OLOP sont sa boucle de planification ouverte et ses bornes optimistes. L'algorithme OLOP cherche donc directement parmi les séquences d'actions celle étant la plus prometteuse vis-à-vis de la borne pour ainsi affiner son estimation de l'action à retourner. Il est à noter que les bornes construites par les algorithmes AMS, OPSS et OLOP ainsi que l'algorithme OP de [Hren and Munos, 2008] sont de vraies bornes supérieures sur la valeur des nœuds ce qui n'est pas le cas pour l'algorithme UCT impliquant ainsi le fait qu'il soit plus risqué à l'utilisation. Cependant, et comme nous l'avons vu précédemment, les résultats de l'algorithme UCT obtenus avec certaines applications sont très bon.

Enfin, comme tout algorithme de planification présenté ici — hormis l'algorithme OPSS — l'algorithme OLOP est indépendant de X mais est influencé par la cardinalité de l'espace d'action A. En effet, plus A est grand plus le facteur de branchement de l'arbre des possibilités est grand faisant ainsi diminuer la précision de l'estimation de la qualité des actions à la racine de par une profondeur plus faible. De plus, ces algorithmes ne sont pas adaptés dans le cas où l'espace d'action est continu à contrario de l'algorithme Hierarchical Optimistic Optimization applied to Trees de [Mansley et al., 2011] que nous allons présenter maintenant.

3.2.7 L'algorithme Hierarchical Optimistic Optimization applied to Trees

[Mansley et al., 2011] propose l'algorithme *Hierarchical Optimistic Optimization applied to Trees* (HOOT) qui est une combinaison de l'algorithme UCT et de l'algorithme *Hierarchical Optimistic Optimization* (HOO) de [Bubeck et al., 2008]. L'algorithme HOO est l'équivalent de l'algorithme UCB1 mais dans le cadre des bandits ayant une infinité de bras appartenant à un espace continu. Cette capacité à sélectionner un bras sur un espace continu est utilisée pour sélectionner en chaque état de l'arbre généré par l'algorithme HOOT, l'action à appliquer en cet état tout comme l'algorithme UCT utilisait l'algorithme UCB1.

L'algorithme HOO procède par discrétisation hiérarchique de l'espace continu des bras $\mathcal{X}$. Cette discrétisation conduit à la création d'un arbre binaire représentant les découpes successives de l'espace continu et sur lequel des bornes sont affixées en chacun de ses nœuds — associées dans le cas présent à un sous-espace de $\mathcal{X}$. La borne est définie ainsi :

$$\begin{aligned} B_{h,i}(n) &= \min\{U_{h,i}(n), \max\{B_{h+1,2i}(n), B_{h+1,2i+1}(n)\}\} \text{ avec} \\ U_{h,i}(n) &= \hat{\mu}_{h,i}(n) + \sqrt{\frac{2\ln n}{N_{h,i}(n)}} + \nu_1 \rho^h \end{aligned}$$

où le couple (h,i) représente un nœud de l'arbre binaire — h pour la profondeur du nœud, i pour son indice — et donc un sous-espace $\mathcal{P}_{h,i}$ de l'espace continu $\mathcal{X}$ de diamètre $\nu_1\rho^h$, $\hat{\mu}_{h,i}(n)$ est la moyenne empirique des récompenses obtenues par les bras appartenant au sous-espace $\mathcal{P}_{h,i}$ jusqu'à l'itération n, $N_{h,i}(n)$ est le nombre de fois qu'un sous-espace inclus dans le sous-espace $\mathcal{P}_{h,i}$ — un enfant du nœud associé donc — est sélectionné pour discrétisation et $0 < \rho < 1$ et $\nu_1 > 0$ sont des paramètres de l'algorithme HOO. L'algorithme HOO est décrit par l'algorithme 3.7. S_n est l'ensemble des nœuds (h,i) de l'arbre pour lesquels $N_{h,i}(n) = 0$, (H_n, I_n) est le nœud sélectionné à l'itération n et les nœuds $(H_n+1, 2I_n)$ et $(H_n+1, 2I_n+1)$ sont les enfants du nœud (H_n, In). Il est à noter que $B_{h,i}(n) = +\infty, \forall (h,i) \in S_n$. Enfin un bras X_n est le bras choisi à l'itération n dans le sous-espace $\mathcal{P}_{H_n,I_n}$ de manière déterministe — le centre du sous-espace par exemple — ou de manière stochastique — tirage uniforme sur le sous-espace par exemple.

La figure 3.6 décrit quatre itérations de l'algorithme HOO. Dans la première itération — voir sous-figure 3.6a —, le nœud racine est sélectionné. Il représente le sous-espace $\mathcal{P}_{0,1}$ qui n'est autre que l'espace continu des bras $\mathcal{X}$ dans son ensemble. Dans notre exemple, un bras X_n est le centre du sous-espace $\mathcal{P}_{H_n,I_n}$. $B_{0,1}(1)$ est mise à jour grâce à la récompense obtenue par l'intermédiaire du bras X_1. Dans la seconde itération représentée par la sous-figure 3.6b, le nœud $(1,2)$ est choisi arbitrairement parmi l'ensemble des nœuds éligibles S_2, les bornes des nœuds étant égales. Une récompense est obtenue par l'intermédiaire du bras $X_2 \in \mathcal{P}_{h,i}$ et les bornes $B_{0,1}(2)$ et

Algorithme 3.7 Hierarchical Optimistic Optimization

function `HOO_init()`
  $n \leftarrow 1$
  Initialiser $\mathcal{S}_n = \{(0,1)\}$

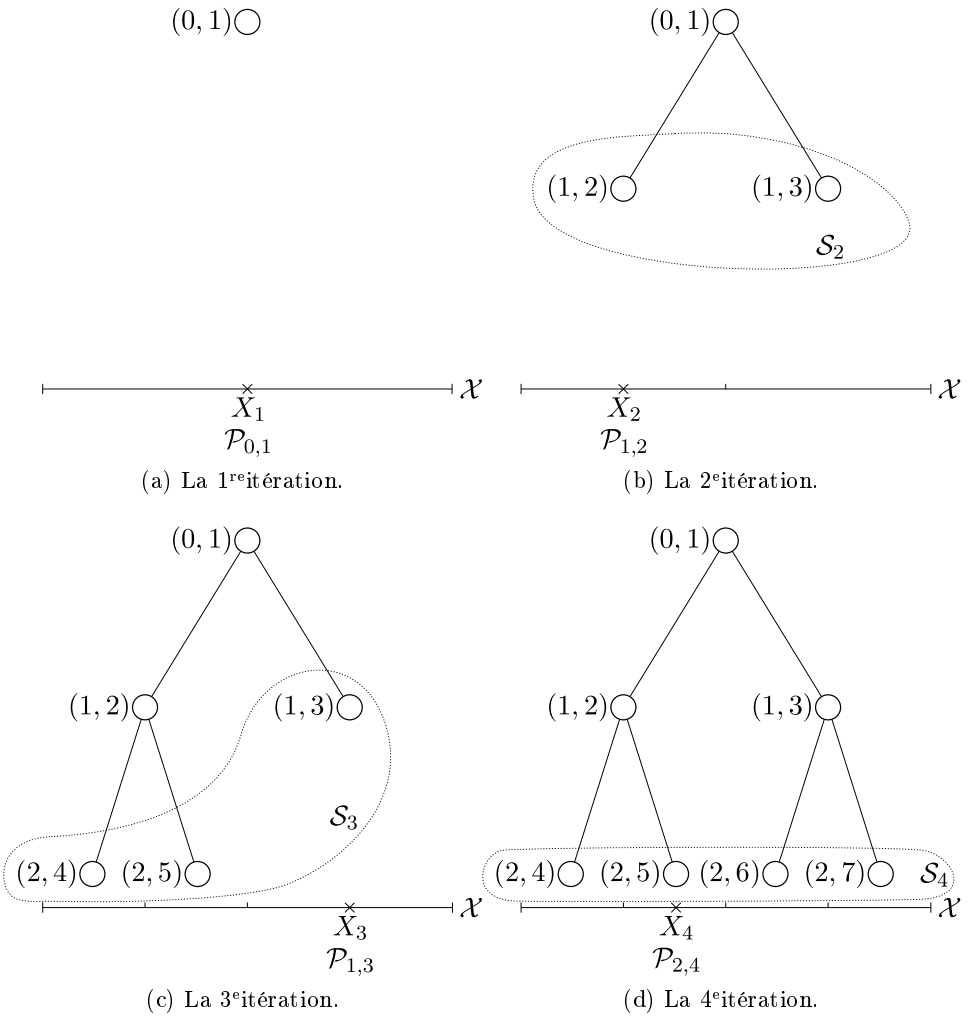
function `HOO_getArm()`
  if n = 1 **then**
    $(H_n, I_n) \leftarrow (0,1)$
  else
    Descendre l'arbre depuis la racine en suivant les bornes $B_{h,i}(n-1)$ maximums pour sélectionner un nœud $(H_n, I_n) \in S_n$
  end if
  $\mathcal{S}_{n+1} \leftarrow \mathcal{S}_n \setminus \{(H_n, I_n)\} \cup \{(H_n + 1, 2I_n), (H_n + 1, 2I_n + 1)\}$
  return Un bras X_n appartenant au sous-espace $\mathcal{X}_{H_n, I_n}$

function `HOO_updateBounds(reward)`
  $r_n \leftarrow$ `reward`
  Mettre à jour les $N_{h,i}(n)$ le long de la trajectoire jusqu'au nœud (H_n, I_n)
  Mettre à jour les bornes $B_{h,i}(n)$
  $n \leftarrow n + 1$

$B_{1,2}(2)$ sont remises à jour. Dans la troisième itération — voir sous-figure 3.6c —, $B_{2,3}(3)$ étant égale à $+\infty$, le nœud $(1,3)$ est sélectionné. Le bras $X_3 \in \mathcal{P}_{1,3}$ est tiré pour obtenir une récompense. Les bornes sont remises à jour. Enfin dans la quatrième itération illustrée par la sous-figure 3.6d, le nœud $(2,5)$ est choisi arbitrairement entre lui-même et le nœud $(2,4)$. Nous pouvons cependant déduire que $B_{1,2}(3) \geq B_{1,3}(4)$. Un bras X_4 est choisi dans le sous-espace $\mathcal{P}_{2,5}$. Le bras est tiré et la récompense obtenue est utilisée pour remettre à jour les bornes. Nous pouvons observer que les mise à jour concernent les nœuds n'appartenant pas à S_n de par la dépendance de B_h, i à n.

Dans son approche, l'algorithme HOO reprend une idée initialement développée par l'algorithme BAST de [Coquelin and Munos, 2007] mais dans un autre contexte et surtout avec des bornes sur le regret.

L'algorithme HOOT utilise l'algorithme HOO pour choisir l'action à effectuer en chaque état d'une séquence simulée. En pratique, l'algorithme HOOT regroupe les états identiques ayant la même profondeur dans l'arbre généré — ou autrement dit ceux étant au même pas de temps dans une séquence. Si l'espace d'états est continu, celui-ci sera discrétisé pour faciliter le regroupement. Les paramètres de l'algorithme HOO ν_1 et ρ seront initialisés respectivement à $\sqrt{d}/2$ et $1/2^d$ avec d le nombre de dimensions de



(a) La 1re itération. (b) La 2e itération.

(c) La 3e itération. (d) La 4e itération.

FIGURE 3.6 – Exemple d'exécution de l'algorithme HOO.

l'espace d'action et donc de l'espace des bras dans l'algorithme HOO. De plus les séquences — et donc l'arbre — seront d'une longueur finie calculée par l'équation suivante :

$$H = \left\lceil \frac{\log\left(\frac{\epsilon(1-\gamma)}{r_{\max}}\right)}{\log(\gamma)} \right\rceil .$$

Enfin l'action retournée par HOOT sera le bras associé à une feuille de l'arbre généré par l'algorithme HOO et associé à l'état x_0. Cette feuille sera au bout d'un chemin partant de la racine et suivant les valeurs $\hat{\mu}_{h,i}(n)$ maximums en chaque nœud. L'algorithme HOOT est décrit par l'algorithme 3.8. G_h est l'ensemble des états rencontrés à la profondeur h. $v_{\max}$ sert ici à normaliser

Algorithme 3.8 Hierarchical Optimistic Optimization applied to Trees

$H \leftarrow \lceil \log\left(\epsilon(1-\gamma)/r_{\max}\right) / \log(\gamma) \rceil$
$G_h \leftarrow \emptyset, \forall H \geq h \geq 0$
repeat
  $h \leftarrow 0$
  repeat
    if $x_h \notin G_h$ **then**
      $G_h \leftarrow G_h \cup \{x_h\}$
      $\texttt{HOO\_init}_{x_h}\texttt{()}$
    end if
    $a \leftarrow \texttt{HOO\_getArm}_{x_h}\texttt{()}$
    $x_{h+1} \sim P(.|x_h, a)$
    $r_h \leftarrow r(x_h, a)$
    $h \leftarrow h + 1$
  until $h > H$
  $h \leftarrow H$
  $q \leftarrow 0$
  repeat
    $q \leftarrow r_h + \gamma q$
    $v_{\max} \leftarrow r_{\max}(1 - \gamma^{H-h+1})/(1-\gamma)$
    $\texttt{HOO\_updateBounds}_{x_h}\texttt{(}q/v_{\max}\texttt{)}$
    $h \leftarrow h - 1$
  until $h < 0$
until Une condition d'arrêt est vraie
return Le bras associé à la feuille du chemin suivi dans l'arbre HOO associé à l'état x_0

la somme des récompenses dépréciées entre 0 et 1 pour chaque état rencontré dans une séquence. En effet, l'algorithme HOO suppose que les récompenses obtenues ont pour support l'intervalle $[0, 1]$.

L'algorithme UCT étant plus efficace quand le nombre d'états successeurs à tout couple état-action est faible, le fait d'avoir des actions continues ne simplifie pas la tâche pour l'algorithme HOOT. C'est pour cela qu'en pratique un espace d'état continu sera discrétisé pour produire un nombre restreint d'états successeurs et permettre ainsi à chaque instance de l'algorithme HOO de discrétiser l'espace d'action de manière suffisante et surtout de recueillir assez d'informations pour faire in fine un choix d'action proche de l'optimal.

Nous allons maintenant décrire un algorithme de planification venant d'un autre domaine : celui de la recherche de plus court chemin.

3.3 Planification et plus court chemin : l'algorithme A*

Dans le domaine de la planification et en particulier de la recherche du plus court chemin dans un graphe, l'algorithme A* [Hart et al., 1968] est l'un des plus connus et a fait l'objet de nombreuses extensions — [Stentz, 1994],[Stentz, 1995],[Koenig et al., 2004a],[Koenig et al., 2004b],[Likhachev et al., 2004],[Koenig and Likhachev, 2006]. Bien que simple dans sa définition, l'algorithme A* est optimal en nombre de nœuds ouverts. Nous allons dans un premier temps définir quelques notions pour ensuite présenter l'algorithme et pouvoir faire dans un second temps un parallèle avec l'algorithme de planification optimiste dans le chapitre suivant.

Soit un graphe G possédant un ensemble de nœuds $\{n_i\}$ contenant éventuellement des informations — dans le cas de la recherche du plus court chemin la position dans l'environnement réel du nœud n_i est une information courante —, un ensemble d'arcs $\{e_{i,j}\}$ dont chaque arc relie un nœud n_i à un nœud n_j mais n'implique pas l'inverse — que le nœud n_j soit relié au nœud n_i — et un coût $c_{i,j}$ associé à chaque arc $e_{i,j}$. Nous supposerons qu'il existe un nœud de départ $s \in \{n_i\}$ définissant un sous graphe G_s des nœuds accessibles depuis s et un ensemble non vide de nœuds finaux T appartenant à G_s et n'incluant pas le nœud s.

Le problème du plus court chemin est de trouver le chemin du nœud s à un nœud $t \in T$ dont la somme des coûts associés aux arcs le constituant soit la plus faible parmi tous les chemins du nœud s à $t, \forall t \in T$. Nous supposerons que le graphe G_s n'est pas disponible dans sa globalité mais au travers d'une fonction successeur Γ qui pour un nœud n_i retourne un ensemble de couples $\{(n_j, c_{i,j})\}$ contenant les nœuds n_j dont il existe un arc $e_{i,j}$ et les coûts associés $c_{j,i}$. Cette représentation est la forme implicite du graphe G_s qui est utilisé dans le cas présent par l'algorithme A* dans sa recherche du plus court chemin.

L'algorithme A* explore le graphe G_s en sélectionnant à chaque itération le nœud à faire évaluer par la fonction successeur Γ pour obtenir de nouveaux nœuds appartenant potentiellement au chemin optimal. Cette sélection s'effectue en choisissant parmi les nœuds marqués «ouverts» le nœud n possédant la valeur $\hat{f}(n)$ la plus basse. Cette valeur $\hat{f}(n)$ est une borne inférieure sur le coût du chemin optimal de s à $t \in T$ passant par n. Cette borne inférieure est définie ainsi :

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n).$$

$\hat{g}(n)$ est l'estimation de la somme des coûts du plus court chemin entre s et n. Cette valeur est remise à jour au gré de l'exploration de G_s par l'algorithme A*. $\hat{h}(n)$ est une borne inférieure sur la somme des coûts entre n et $t \in T$. Cette borne inférieure doit être consistante sur l'ensemble des nœuds de G_s

ou de manière formelle, elle doit vérifier l'inégalité suivante :

$$h(m,n) + \hat{h}(n) \geq \hat{h}(m) \tag{3.3}$$

pour tous nœuds m et n appartenant à G_s. Un nœud «ouvert» est un nœud ayant été retourné par un appel à Γ mais n'ayant pas encore été sélectionné pour être évalué par la fonction Γ. À l'inverse, un nœud évalué par la fonction Γ sera dit «fermé». Il est à noter que $\hat{g}(n) = g(n)$ si n est «fermé». Si le nœud sélectionné pour être évalué appartient à T alors l'algorithme se termine et le plus court chemin a été trouvé. L'algorithme A* est décrit par l'algorithme 3.9.

Algorithme 3.9 A*

Marquer s comme étant «ouvert»
Calculer $\hat{f}(s)$
repeat
  Sélectionner le nœud «ouvert» n dont la valeur $\hat{f}(n)$ est la plus faible
  Marquer n comme étant «fermé»
  Évaluer $\Gamma(n)$
  Marquer les nœuds successeurs de n comme étant «ouvert» sauf ceux déjà marqués comme étant «fermé»
  Calculer $\hat{f}$ pour chacun des nœuds successeurs de n
until $n \in T$

Dans le chapitre suivant nous présenterons un nouvel algorithme de planification dans le cadre des MDPs déterministes utilisant une approche optimiste et ayant des liens de parentés avec l'algorithme A* que nous établirons.

Chapitre 4

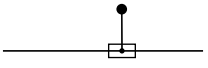
Planification Optimiste

Dans ce chapitre, nous présentons notre première contribution, l'algorithme de Planification Optimiste — *Optimistic Planning* en anglais. Nous allons dans un premier temps décrire le cadre dans lequel s'inscrit cet algorithme ainsi que la notion de regret qui lui est associée. Nous décrirons ensuite l'algorithme de planification uniforme et nous analyserons son regret que nous mettrons en relation avec celui de l'algorithme de planification optimiste dans une troisième partie le décrivant et l'analysant ainsi que faisant le lien avec l'algorithme A*. Enfin nous présenterons différentes expérimentations pour mettre en parallèle les résultats théoriques et pratiques.

4.1 Planification sous contrainte de ressources computationnelles finies

Comme présentée dans le chapitre précédent, la planification dans les processus de décision markoviens consiste à trouver l'action à appliquer dans l'état courant x_{crt} du système à contrôler et ce dans l'objectif d'optimiser la somme future des récompenses. Nous nous intéressons ici en particulier au cas où les ressources computationnelles disponibles pour effectuer la planification ne sont pas connues à l'avance. Ces ressources computationnelles peuvent se caractériser sous la forme d'un nombre limité d'appels au modèle génératif associé au système à contrôler, d'un espace mémoire restreint pour l'exécution de l'algorithme ou encore d'un temps processeur limité par la disponibilité du dit processeur ou par une contrainte de temps lié au système à contrôler. En effet il n'est pas rare que le laps de temps disponible entre deux prises d'action sur un système à contrôler soit contraint mais aussi et surtout inconnu.

Les ressources computationnelles disponibles étant finies et inconnues, l'algorithme de sélection d'action considéré devra alors être anytime et ainsi pouvoir fournir une action qui tendra vers l'action optimale au fur et à mesure que des ressources computationnelles lui seront attribuées. Nous supposerons



que le système à contrôler est déterministe et modélisé par un processus de décision markovien dont nous possédons le modèle génératif $f : X \times A \to X$ avec A un espace d'action discret et X un espace d'état quelconque — discret, continu, infini, ... — tel que $x_{t+1} = f(x_t, a_t)$, x_{t+1} étant l'état successeur lorsque l'action a_t est effectuée dans l'état x_t. Nous aurons aussi accès à la fonction récompense $r : X \times A \to \mathbb{R}$ telle que $r_t = r(x_t, a_t)$.

Comme vu précédemment, l'utilisation successive de l'algorithme de sélection d'action sur l'état courant x_{crt} du système à contrôler définit une politique déterministe $\pi_{\mathcal{A}}$ avec $\pi_{\mathcal{A}}(x) = \mathcal{A}(n)$ avec $\mathcal{A}(n)$ l'action retournée par l'algorithme de sélection d'action $\mathcal{A}$ en l'état x après l'utilisation de n ressources computationnelles.

Pour quantifier la performance de la politique $\pi_{\mathcal{A}}$ engendrée par un algorithme de sélection d'action $\mathcal{A}$, il convient de définir en premier la notion de regret. Le regret d'un algorithme de sélection d'action $\mathcal{A}$ est la différence de qualité entre l'action optimale et l'action retournée par l'algorithme pour un état x. Ainsi le regret d'un algorithme de sélection d'action $\mathcal{A}$ retournant une action $\mathcal{A}(n)$ après l'utilisation de n ressources computationnelles peut s'exprimer ainsi :

$$R_{\mathcal{A}}(n) \stackrel{\text{def}}{=} \max_{a \in A} Q^*(x, a) - Q^*(x, \mathcal{A}(n)). \tag{4.1}$$

Nous pouvons ainsi facilement voir qu'un algorithme de sélection d'action $\mathcal{A}$ cherchant à minimiser son regret devrait produire une politique $\pi_{\mathcal{A}}$ proche de l'optimale.

Proposition 1. *Considérons un algorithme de sélection d'action $\mathcal{A}$ avec un regret ϵ — c'est à dire que pour chaque état x, l'algorithme de sélection d'action retourne une action a telle que $V^*(x) - Q^*(x, a) \leq \epsilon$. Alors la performance de la politique $\pi_{\mathcal{A}}$ ainsi définie est $\frac{\epsilon}{1-\gamma}$ optimale, c'est-à-dire que pour tout x :*

$$V^*(x) - V^{\pi_{\mathcal{A}}}(x) \leq \frac{\epsilon}{1-\gamma}$$

avec $\gamma \in [0, 1[$ le facteur de dépréciation.

Démonstration. Nous supposons que

$$\epsilon \geq V^*(x) - Q^*(x, \mathcal{A}(x))$$

donc

$$\begin{aligned}
\epsilon &\geq V^*(x_t) - (r_t + \gamma V^*(x_{t+1})) \\
&\geq V^*(x_t) - (r_t + \gamma(Q^*(x_{t+1}, \mathcal{A}(x_{t+1})) + \epsilon) \\
&\geq V^*(x_t) - (r_t + \gamma Q^*(x_{t+1}, \mathcal{A}(x_{t+1}))) - \gamma\epsilon \\
&\geq V^*(x_t) - (r_t + \gamma(r_{t+1} + \gamma V^*(x_{t+2})) - \gamma\epsilon \\
&\geq V^*(x_t) - (r_t + \gamma r_{t+1} + \gamma^2 Q^*(x_{t+2}, \mathcal{A}(x_{t+2})) - (\gamma\epsilon + \gamma^2\epsilon).
\end{aligned}$$

En continuant à développer $Q^*(x_t, \mathcal{A}(x_t))$, nous obtenons :

$$\epsilon + \gamma\epsilon + \gamma^2\epsilon + \cdots \geq V^*(x_t) - V^{\pi_\mathcal{A}}(x_t).$$

Enfin γ appartenant à $[0, 1[$, nous pouvons conclure que :

$$\frac{\epsilon}{1-\gamma} \geq V^*(x_t) - V^{\pi_\mathcal{A}}(x_t)$$

□

Comme dit précédemment, un algorithme de sélection d'action $\mathcal{A}$ cherchant à réduire son regret $R_\mathcal{A}(n)$ améliorera aussi la performance de la politique $\pi_\mathcal{A}$ qu'il engendre. L'algorithme de planification uniforme — présenté dans la section suivante — voit son regret $R_{\mathcal{A}_U}(n)$ borné — borne supérieure et inférieure — par $\Theta(n^{-\frac{\log 1/\gamma}{\log K}})$ avec K le facteur de branchement de l'arbre des possibilités — typiquement $K = |A|$. Il est intéressant de remarquer que la borne est indépendante de la dimension de l'espace d'état mais est cependant dépendante du facteur de branchement K et du facteur de dépréciation γ.

Comme le suggère [Péret and Garcia, 2004], une exploration non-uniforme de l'arbre des possibilités devrait permettre d'obtenir une meilleure borne supérieure sur le regret par rapport à celle de l'algorithme de planification uniforme et ainsi obtenir une politique plus performante tout en restant indépendante de la malédiction de la dimension.

L'algorithme présenté dans ce chapitre se propose d'explorer les feuilles les plus prometteuses en premier au travers d'une borne supérieure sur la somme des récompenses dépréciées accessibles depuis une feuille et ce pour chaque feuille. Ce principe n'est pas sans rappeler celui de l'algorithme UCB de [Auer et al., 2002] présenté précédemment. De plus cette approche est voisine de celles développées par l'algorithme BAST de [Coquelin and Munos, 2007] ou encore par l'algorithme HOO de [Bubeck et al., 2008] dans le sens où la régularité implicite de l'arbre des possibilités — provenant du fait que l'on considère la somme des récompenses dépréciées le long des chemins explorés — est utilisée pour définir une borne supérieure sur ses nœuds.

Nous allons maintenant définir l'algorithme de planification uniforme qui servira de repère dans la comparaison entre son regret et celui de la solution proposée dans ce chapitre.

4.2 Planification Uniforme

4.2.1 Description

Étant donné un état x, nous allons décrire une méthode pour sélectionner une action presque optimale basée sur la construction d'un arbre des possibilités uniforme. Nous considérons l'arbre — infini — $\mathcal{T}$ composé de tous les états atteignables depuis l'état x : la racine correspond à x et chaque nœud de profondeur d correspond à un état qui est atteignable depuis x après une séquence de d transitions. Chaque nœud i — associé à un état x_t — a K enfants $\mathcal{C}(i)$ — associés aux états $\{x_{t+1} = f(x_t, a)\}_{a \in A}$. Écrivons 0 le nœud racine et $1 \dots K$ ses K enfants — les nœuds de profondeur 1.

Nous appelons un chemin dans $\mathcal{T}$, une séquence — finie ou infinie — de nœuds connectés partant de la racine. Nous définissons la valeur v_i d'un nœud i comme le majorant sur tous les chemins infinis passant par le nœud i de la somme des récompenses dépréciées obtenues le long d'un de ces chemins. Nous avons la propriété que $v_i = \max_{h \in \mathcal{C}(i)} v_j$ et donc que la valeur optimale — la valeur de la racine — $V^*(x_{\text{crt}}) = v^* = v_0 = \max_{i \in \mathcal{T}} v_i = \max\{v_1, ..., v_K\}$.

Nous considérons les ressources computationnelles comme étant exprimées en nombre de nœuds étendus. Ceci se traduit directement en temps processeur nécessaire pour explorer la partie correspondante de l'arbre ou la quantité de mémoire requise pour stocker les informations issues des nœuds étendus — les états et récompenses associés typiquement. Ceci est aussi équivalent au nombre d'appels au modèle génératif $f(x_t, a)$ fournissant l'état successeur x_{t+1}.

Nous disons qu'un nœud est étendu si des ressources computationnelles ont été allouées à ce nœud ainsi qu'au calcul des transitions vers ses enfants — par le biais d'un appel au modèle génératif pour chaque action. À chaque itération n, l'arbre des nœuds étendus $\mathcal{T}_n$ dénote l'ensemble des nœuds ayant été étendus. L'ensemble des nœuds appartenant à $\mathcal{T}$ qui ne sont pas dans $\mathcal{T}_n$ mais dont les parents sont dans $\mathcal{T}_n$ est dénommé $\mathcal{S}_n$: ceci représente l'ensemble des nœuds éligibles à être étendus pendant l'itération suivante — voir la figure 4.1.

Pour tout nœud $i \in \mathcal{S}_n$, nous définissons la valeur u_i comme étant la somme des récompenses dépréciées obtenues le long du chemin — fini — depuis la racine jusqu'au nœud i — cette information est disponible à l'itération n car les parents de i ont été étendus et donc un appel au modèle génératif a été effectué pour obtenir le nœud i. Maintenant, pour tout nœud $i \in \mathcal{T}_n$ nous définissons de manière récursive $u_i = \max_{j \in \mathcal{C}(i)} u_j$. La valeur u étant définie sur $\mathcal{S}_n$, elle est aussi bien définie sur $\mathcal{T}_n$.

Notons que puisque la valeur u dépend de $\mathcal{T}_n$, nous la dénoterons $u_i(n)$ à chaque occasion où cette dépendance vis-à-vis de n est importante. De par sa définition, nous avons la propriété que $u_i(n)$ est une fonction croissante

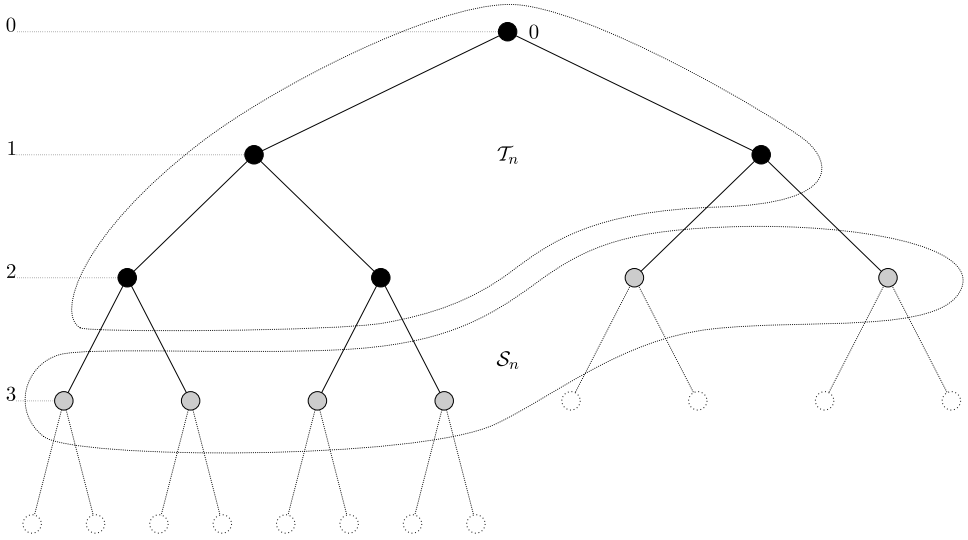


FIGURE 4.1 – L'ensemble des nœuds étendus $\mathcal{T}_n$ — disque noir — à l'itération $n = 5$ et l'ensemble des nœuds $\mathcal{S}_n$ — disque gris — éligibles à être étendus à l'itération suivante. Nous sommes dans le cas où $K = 2$.

de n.

Nous supposons maintenant que toute récompense obtenue au travers du modèle génératif appartient à l'intervalle $[0, 1]$. Ainsi puisque la somme des récompenses dépréciées depuis un nœud de profondeur d est au plus $\gamma^d + \gamma^{d+1} + \cdots = \frac{\gamma^d}{1-\gamma}$ avec $\gamma \in [0, 1[$, nous avons pour tout $i \in \mathcal{T}_n \cup \mathcal{S}_n$ et $n \geq t \geq 1$:

$$u_i(n) \leq v_i \leq u_i(n) + \frac{\gamma^d}{1-\gamma}.$$

De ces définitions, nous décrivons maintenant une politique d'exploration uniforme de l'arbre infini des possibilités. Dans un premier temps, le nœud racine est étendu. Puis à chaque itération n, nous étendons un nœud appartenant à $\mathcal{S}_n$ possédant la profondeur la plus faible — voir l'algorithme 4.1.

Algorithme 4.1 Planification Uniforme $\mathcal{A}_U$

Étendre le nœud racine
$n \leftarrow 1$
while Une condition d'arrêt n'est pas vraie **do**
  Étendre un nœud $i \in \mathcal{S}_n$ ayant la profondeur la plus faible
  $n \leftarrow n + 1$
end while
return L'action $k \in \{1, \ldots, K\}$ qui maximise $u_k(n)$

De part cette définition, à chaque itération l'arbre étendu est uniforme. Ainsi à l'itération $n = 1 + K + K^2 + \cdots + K^d = \frac{K^{d+1}-1}{K-1}$, tous les nœuds de profondeur d ou moins ont été étendus. L'algorithme de sélection d'action uniforme dénommé $\mathcal{A}_U$ retourne après n itérations l'action correspondant à l'enfant du nœud racine $k \in \{1, \ldots, K\}$ possédant la valeur $u_k(n)$ la plus élevée, c'est à dire que :

$$\mathcal{A}_U(n) \stackrel{\text{def}}{=} \arg\max_{k \in \{1,\ldots,K\}} u_k(n),$$

les égalités étant résolues arbitrairement. Autrement dit, après n itérations pendant lesquelles l'arbre fut étendu uniformément, $\mathcal{A}_U$ sélectionne l'action correspondant à la branche menant au chemin — fini — possédant la somme des récompenses dépréciées la plus élevée.

Nous allons maintenant analyser le regret de l'algorithme de sélection d'action uniforme décrit à l'instant.

4.2.2 Analyse

Théorème 1. *Considérons l'algorithme de planification uniforme décrit précédemment. Alors, pour toute fonction récompense, le regret de l'algorithme de planification uniforme est borné par*

$$R_{\mathcal{A}_U}(n) \leq \frac{1}{\gamma(1-\gamma)} \left[n(K-1)+1\right]^{-\frac{\log 1/\gamma}{\log K}}.$$

De plus, pour tout $n \geq 3$, il existe une fonction récompense telle que le regret de cet algorithme est au moins

$$R_{\mathcal{A}_U} \geq \frac{\gamma}{1-\gamma} \left[n(K-1)+1\right]^{-\frac{\log 1/\gamma}{\log K}}. \tag{4.2}$$

Nous déduisons alors la dépendance suivante — dans le sens du pire des cas :

$$R_{\mathcal{A}_U}(n) = \Theta\left(n^{-\frac{\log 1/\gamma}{\log K}}\right).$$

Démonstration. Pour tout $n \geq 3$, soit d le plus grand entier tel que

$$n \geq \frac{K^{d+1}-1}{K-1}. \tag{4.3}$$

Ainsi tous les nœuds de profondeur d ont été étendus. De plus, pour tout $k \in \{1, \ldots, K\}$, nous avons $v_k \leq u_k + \frac{\gamma^{d+1}}{1-\gamma}$ car toutes les récompenses jusqu'à la profondeur d ont été obtenues. Ainsi :

$$\begin{array}{rcl} v^* & = & \max_{k\in\{1,\ldots,K\}} v_k \\ & \leq & \max_{k\in\{1,\ldots,K\}} u_k + \frac{\gamma^{d+1}}{1-\gamma} \quad = \quad u_{\mathcal{A}_U(n)} + \frac{\gamma^{d+1}}{1-\gamma} \\ & & \leq \quad v_{\mathcal{A}_U(n)} + \frac{\gamma^{d+1}}{1-\gamma}. \end{array}$$

Maintenant à partir de l'équation (4.3) et comme d est le plus grand entier tel que l'équation (4.3) est satisfaite, nous avons alors $d+1 > \log_K[n(K-1)+1]-1$. Ainsi toutes les récompenses à partir de la profondeur $d+1$ n'ayant pas été observées, nous pouvons déduire que

$$\begin{aligned} v^* - v_{\mathcal{A}_U(n)} &\leq \frac{\gamma^{d+1}}{1-\gamma} \\ &\leq \frac{\gamma^{d+2}}{\gamma(1-\gamma)} \\ &\leq \frac{1}{\gamma(1-\gamma)}\gamma^{\log_K[n(K-1)+1]} \\ &\leq \frac{1}{\gamma(1-\gamma)}[n(K-1)+1]^{\log_K \gamma} \\ &\leq \frac{1}{\gamma(1-\gamma)}[n(K-1)+1]^{-\frac{\log 1/\gamma}{\log K}}. \end{aligned}$$

Pour la deuxième partie du théorème, considérons une valeur de n fixée. Définissons d comme étant l'entier le plus grand tel que l'équation (4.3) est satisfaite. Ainsi à partir de (4.3), nous avons $d \leq \log_K[n(K-1)+1]-1$. Définissons la fonction récompense suivante : toutes les récompenses sont égales à 0 sauf dans une branche — la branche 1 par exemple — où les récompenses obtenues pour les transitions des nœuds de profondeur p aux nœuds de profondeur $p+1$ pour $p > d+1$ sont égales à 1 — $p > d+1$ car il peut exister des nœuds étendus de profondeur $d+1$, voir Figure 4.2. De fait $v^* = v_1 = \frac{\gamma^{d+2}}{1-\gamma}$ et $v_2 = 0$.

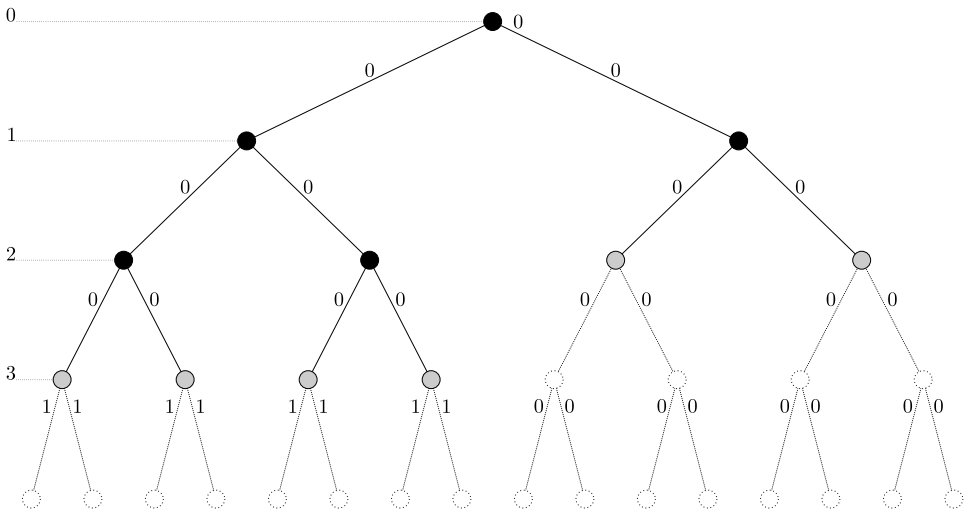


FIGURE 4.2 – Dans cette exemple, $n = 5$ donc $d = 1$. Les récompenses sont représentées à côté des branches de l'arbre et sont égales à 0 partout sauf dans la branche 1 où les récompenses des transitions d'un état de profondeur p à un état de profondeur $p+1$ avec $p > d+1$ sont égales à 1.

Ainsi au moment de la décision, tous les u_k pour $k \in \{1, \ldots, K\}$ sont égaux à 0 — seules les récompenses égales à 0 ont été observées — et donc

une action est sélectionnée arbitrairement par l'algorithme — l'action 2 par exemple.

Par conséquent $v^* - v_{\mathcal{A}_U} = v_1 - v_2 = \frac{\gamma^{d+2}}{1-\gamma}$ et ainsi l'équation (4.2) est dérivée depuis la borne sur d. □

Dès lors, grâce au théorème 1 et à la proposition 1, nous pouvons déduire que pour garantir que la performance de la politique $\pi_{\mathcal{A}_U}$ dérivée de l'algorithme de planification uniforme est ϵ-optimale — c'est à dire que $\|V^* - V^{\pi_{\mathcal{A}_U(n)}}\|_\infty \leq \epsilon$ —, nous devons allouer $n = \Theta(\frac{1}{\epsilon(1-\gamma)^2})^{\frac{\log K}{\log 1/\gamma}}$ ressources computationnelles par prise de décision.

Notons que l'analyse du pire cas peut nous décourager dans la recherche de meilleures bornes car une analyse similaire pourrait être appliquée sur tout algorithme de la même famille. Cependant nous pouvons définir des classes spécifiques de problèmes dans lesquelles nous pourrions espérer obtenir de meilleurs taux de convergence — pour le regret — en utilisant d'autres algorithmes de sélection d'action que l'uniforme : nous ne pouvons pas espérer obtenir de meilleurs taux pour tous les problèmes mais cela peut être possible pour des classes spécifiques de problèmes.

Dans la section suivante, nous considérons une approche optimiste de la planification.

4.3 Planification Optimiste

Dans cette section, nous présentons un algorithme construisant un arbre des possibilités asymétrique en explorant en premier les parties de l'arbre les plus prometteuses.

4.3.1 Description

Nous définissons dans un premier temps la valeur b pour tout nœud $i \in \mathcal{S}_n$ de profondeur d telle que $b_i \stackrel{\text{def}}{=} u_i + \frac{\gamma^d}{1-\gamma}$ et, pour tout nœud $i \in \mathcal{T}_n$, b est définie récursivement telle que $b_i \stackrel{\text{def}}{=} \max_{j \in \mathcal{C}(i)} b_j$.

Notons que comme la valeur u, la valeur b est bien définie et dépend de n. Ainsi elle sera dénotée $b_i(n)$ lorsque la dépendance explicite en n sera importante. Une propriété de $b_i(n)$ est d'être une fonction décroissante de n. De plus, puisque les récompenses sont comprises dans l'intervalle $[0, 1]$, nous avons immédiatement la propriété que la valeur b est une borne supérieure sur la valeur des nœuds : pour tout $i \in \mathcal{T}_t \cup \mathcal{S}_t$, pour tout $n \geq t$,

$$u_i(n) \leq v_i \leq b_i(n).$$

La politique d'exploration optimiste présentée ici consiste à choisir à chaque itération un nœud $i \in \mathcal{S}_n$ possédant la plus grande valeur b — voir l'algorithme 4.2. L'action retournée correspond à l'enfant du nœud racine ayant la plus grande valeur u telle que

$$\mathcal{A}_O(n) \stackrel{\text{def}}{=} \arg\max_{k \in \{1 \ldots K\}} u_k(n),$$

les égalités étant résolues arbitrairement.

Algorithme 4.2 Planification Optimiste $\mathcal{A}_O$

Étendre le nœud racine
$n \leftarrow 1$
while Une condition d'arrêt n'est pas vraie **do**
  Étendre un nœud $i \in \mathcal{S}_n$ tel que $\forall j \in \mathcal{S}_n, b_i(n) \geq b_j(n)$
  $n \leftarrow n + 1$
end while
return L'action $k \in \{1, \ldots, K\}$ qui maximise $u_k(n)$

Nous allons maintenant analyser le regret de l'algorithme de planification optimiste et décrire une classe de problèmes dans laquelle cet algorithme possède une meilleure borne supérieure sur le regret que l'algorithme de planification uniforme.

4.3.2 Analyse

Théorème 2. *Considérons l'algorithme de planification optimiste décrit précédemment. À l'itération n, le regret est borné par*

$$R_{\mathcal{A}_O}(n) \leq \frac{\gamma^{d_n}}{1-\gamma} \tag{4.4}$$

où d_n est la profondeur de l'arbre des nœuds étendus $\mathcal{T}_n$ — la profondeur maximale des nœuds dans $\mathcal{T}_n$.

Par conséquent, pour toute fonction récompense, la borne supérieure sur le regret pour l'algorithme de planification optimiste n'est pas plus large que celle de l'algorithme de planification uniforme. En effet, puisque la stratégie d'exploration de l'algorithme de planification uniforme est de minimiser d_n, la profondeur obtenue par l'utilisation de l'algorithme de planification optimiste est au moins aussi grande que celle obtenue par l'algorithme de planification uniforme.

Démonstration. Notons en premier que l'action retournée par l'algorithme de planification optimiste correspond à un chemin conduisant à l'un des nœuds les plus profonds de l'arbre $\mathcal{T}_n$. Dans le cas contraire, pour $i \in \mathcal{T}_n$ un nœud de profondeur maximale d_n, il existerait un nœud $j \in \mathcal{T}_n$ de profondeur $d < d_n$ tel que $u_j(n) \geq u_i(n)$. Ainsi il existerait une itération t pendant laquelle le nœud $i \in \mathcal{S}_t$ a été étendu avec $t < n$ si le nœud j n'avait pas encore été étendu à l'itération $n-1$ ou $t \leq n$ dans le cas contraire. De fait, nous définissons un nœud $k \in \mathcal{T}_t \cup \mathcal{S}_t$ tel que

$$k = \begin{cases} j & \text{si } j \in \mathcal{T}_t \cup \mathcal{S}_t \\ \text{un nœud parent de } j & \text{sinon.} \end{cases}$$

Le nœud i étant étendu à l'itération t, $b_i(t) \geq b_k(t) \geq b_k(n) \geq b_j(n)$. Mais c'est impossible car

$$\begin{aligned} b_i(t) &= u_i(t) + \tfrac{\gamma^{d_n}}{1-\gamma} \\ &\leq u_i(n) + \tfrac{\gamma^{d_n}}{1-\gamma} \\ &\leq u_j(n) + \tfrac{\gamma^{d_n}}{1-\gamma} \\ &< u_j(n) + \tfrac{\gamma^{d}}{1-\gamma} \\ &< b_j(n). \end{aligned}$$

Par conséquent, l'action retournée par l'algorithme correspond au chemin qui a été le plus profondément exploré.

Soit $i \in \mathcal{T}_n$ un nœud de profondeur maximale d_n. Le nœud i appartient à l'une des K branches connectées à la racine, supposons que celle-ci soit la branche 1. Si l'action optimale est 1 alors le regret est de 0 et la borne supérieure est satisfaite. Sinon l'action optimale n'est pas 1 mais 2 par exemple. Soit $t \leq n$ l'itération à laquelle i a été étendu. Nous avons donc

$b_i(t) \geq b_j(t)$ pour tout nœud $j \in \mathcal{S}_t$ mais aussi pour tout nœud $j \in \mathcal{T}_t$. En particulier, $b_1(t) = b_i(t) \geq b_2(t)$. Cependant $b_2(t) \geq b_2(n)$. Nous savons que la valeur u est une borne inférieure sur la valeur v ainsi $u_1(t) \leq v_1$ et de par la définition de la valeur u, $u_1(t) \leq u_1(n)$. Par conséquent nous avons :

$$\begin{aligned} v^* - v_{\mathcal{A}_O}(n) &= v_2 - v_1 \\ &\leq b_2(n) - v_1 \\ &\leq b_2(t) - v_1 \\ &\leq b_1(t) - u_1(t) \\ &\leq b_i(t) - u_i(t) = \frac{\gamma^{d_n}}{1-\gamma}. \end{aligned}$$

□

Remarque 1. Ce résultat montre que la borne supérieure sur le regret pour l'algorithme de planification optimiste ne peut pas être pire que celle pour l'algorithme de planification uniforme. Cela ne veut pas dire cependant, que pour tout problème, l'algorithme de planification optimiste offrira d'aussi bonnes performances que l'algorithme de planification uniforme. En effet, car si nous considérons l'exemple mentionné dans la preuve du théorème 1, où à l'itération n toutes les récompenses observées sont égales à 0, alors tout algorithme — y compris l'optimiste — retournera une action choisie arbitrairement laquelle pourrait se révéler être pire qu'une autre action choisie arbitrairement — par l'algorithme uniforme par exemple. Cependant, si nous considérons des classes d'équivalences de problèmes, où les classes sont définies par des arbres possédant la même fonction récompense moyennant de possibles permutations de branches, alors nous conjecturons que nous avons un résultat plus fort tel que pour tout problème, l'algorithme de planification optimiste n'est jamais pire que l'algorithme de planification uniforme, et ce lorsqu'ils sont confrontés sur un problème de la même classe.

Notons que la borne inférieure sur le regret obtenue pour l'algorithme de planification uniforme est aussi valable pour l'algorithme de planification optimiste — la preuve est la même : puisque jusqu'à l'itération n toutes les récompenses obtenues sont égales à 0, l'algorithme de planification optimiste construira un arbre uniforme. Ceci montre qu'aucune amélioration par rapport à l'algorithme de planification uniforme ne peut être espérée dans le pire des cas comme cela fut déjà mentionné précédemment. Dans l'optique de quantifier les améliorations possibles par rapport à l'algorithme de planification uniforme, nous allons définir des classes de problèmes qui feront l'objet du reste de l'analyse.

Pour tout $\epsilon \in [0, 1]$, nous définissons la proportion $p(\epsilon)$ de chemins ϵ-optimaux comme étant la limite, quand $d \to \infty$, de la proportion $p_d(\epsilon)$ de nœuds de profondeur d ϵ-optimaux, c'est à dire

$$p(\epsilon) \stackrel{\text{def}}{=} \lim_{d \to \infty} \frac{|\{\text{nœud } i \text{ de profondeur } d \text{ tel que } v^* - v_i \leq \epsilon\}|}{K^d}.$$

Notons que cette limite est bien définie car $p_d(\epsilon)$ est une fonction décroissante de d.

Théorème 3. *Soit $\beta \in [0, \frac{\log K}{\log 1/\gamma}]$ tel que la proportion de chemins ϵ-optimaux dans l'arbre $\mathcal{T}$ est $O(\epsilon^\beta)$. Définissons $\kappa \stackrel{\text{def}}{=} K\gamma^\beta$ lequel appartient à l'intervalle $[1, K]$.*

Si $\beta < \frac{\log K}{\log 1/\gamma}$ — c'est-à-dire que $\kappa > 1$ —, alors le regret de l'algorithme de planification optimiste est

$$R_{\mathcal{A}_O}(n) = O\left(n^{-\frac{\log 1/\gamma}{\log \kappa}}\right).$$

Si $\beta = \frac{\log K}{\log 1/\gamma}$ — c'est-à-dire que $\kappa = 1$ —, soit c une constante telle que $p(\epsilon) \leq c\epsilon^\beta$. Alors nous avons le taux exponentiel :

$$R_{\mathcal{A}_O}(n) = O\left(\gamma^{\frac{(1-\gamma)^\beta n}{c'}}\right),$$

où c' est une constante strictement plus grande que c.

Démonstration. Définissons le sous-arbre $\mathcal{T}_\infty \subset \mathcal{T}$ comme l'ensemble des nœuds i de profondeur d étant $\frac{\gamma^d}{1-\gamma}$-optimaux, c'est-à-dire

$$\mathcal{T}_\infty \stackrel{\text{def}}{=} \bigcup_{d \geq 0} \left\{ \text{nœud } i \text{ de profondeur } d \text{ tel que } v_i \geq v^* - \frac{\gamma^d}{1-\gamma} \right\}.$$

Prouvons maintenant que les nœuds étendus par l'algorithme de planification optimiste appartiennent à $\mathcal{T}_\infty$. En effet, soit i un nœud de profondeur d étendu à l'itération n. Alors pour tout $j \in \mathcal{T}_n \cup \mathcal{S}_n$, $b_i(n) \geq b_j(n)$. Ainsi $b_i(n) = b_0(n)$ avec b_0 la valeur b de la racine. Mais $b_0(n) \geq v_0 = v^*$, ainsi

$$\begin{array}{rcccl} v_i & \geq & u_i(n) & = & b_i(n) - \frac{\gamma^d}{1-\gamma} \\ & & & \geq & v^* - \frac{\gamma^d}{1-\gamma}, \end{array}$$

donc $i \in \mathcal{T}_\infty$.

Maintenant, à partir de la définition de β, il existe d_0 tel que la proportion de nœuds de profondeur $d > d_0$ ϵ-optimaux est au plus $c\epsilon^\beta$, où c est une constante. Nous avons donc le nombre n_d de nœuds de profondeur d dans $\mathcal{T}_\infty$ borné par $c(\frac{\gamma^d}{1-\gamma})^\beta K^d$.

Soit d_n la profondeur de l'arbre des nœuds étendus $\mathcal{T}_n$ à l'itération n. Soit $n_0 = \frac{K^{d_0+1}-1}{K-1}$ le nombre de nœuds dans $\mathcal{T}$ de profondeur plus faible ou égale à d_0. Nous avons :

$$\begin{aligned} n &\leq n_0 + \sum_{d=d_0+1}^{d_n} n_d \\ &\leq n_0 + c \sum_{d=d_0+1}^{d_n} \left(\frac{\gamma^d}{1-\gamma}\right)^\beta K^d = n_0 + c' \sum_{d=d_0+1}^{d_n} \kappa^d \end{aligned}$$

avec $c' = c/(1-\gamma)^\beta$ et $\kappa = \gamma^\beta K$.

En premier lieu, si $\kappa > 1$ alors nous avons :

$$\begin{aligned} n &\leq n_0 + c' \tfrac{\kappa^{d_0+1} - \kappa^{d_n+1}}{1-\kappa} \\ n &\leq n_0 + c' \tfrac{\kappa^{d_n+1} - \kappa^{d_0+1}}{\kappa - 1} \\ \tfrac{(n-n_0)(\kappa-1)}{c'} + \kappa^{d_0+1} &\leq \kappa^{d_0+1} \\ \log_\kappa \left[\tfrac{(n-n_0)(\kappa-1)}{c'} + \kappa^{d_0+1} \right] - 1 &\leq d_n \\ \log_\kappa \left[\tfrac{(n-n_0)(\kappa-1)}{c'\kappa} + \kappa^{d_0} \right] &\leq d_n. \end{aligned}$$

Par conséquent et grâce au théorème 2, nous avons le regret :

$$\begin{aligned} R_{\mathcal{A}_O}(n) &\leq \tfrac{\gamma^{d_n}}{1-\gamma} \\ &\leq \tfrac{1}{1-\gamma} \left[\tfrac{(n-n_0)(\kappa-1)}{c'\kappa} + \kappa^{d_0} \right]^{\log_\kappa \gamma} \\ &= O\left(n^{-\frac{\log 1/\gamma}{\log \kappa}} \right). \end{aligned}$$

En second lieu, si $\kappa = 1$ et si $p(\epsilon) \leq c\epsilon^\beta$ avec c une constante alors, pour tout $c' > c$, il existe d_0 tel que pour tout $d > d_0$, $p_d(\epsilon) \leq c'\epsilon^\beta$. Soit $n_0 = \frac{K^{d_0+1}-1}{K-1}$ le nombre de nœuds dans $\mathcal{T}$ de profondeur plus faible ou égale à d_0. En utilisant les mêmes arguments que précédemment, nous déduisons que $n \leq n_0 + \frac{c'}{(1-\gamma)^\beta}(d_n - d_0)$ et donc que le regret :

$$\begin{aligned} R_{\mathcal{A}_O}(n) &\leq \tfrac{\gamma^{d_n}}{1-\gamma} \\ &= O\left(\gamma^{n \frac{(1-\gamma)^\beta}{c'}} \right). \end{aligned}$$

□

Remarque 2. Il est à noter qu'une valeur de β satisfaisant $p(\epsilon) = O(\epsilon^\beta)$ se situe nécessairement dans l'intervalle $[0, \frac{\log K}{\log 1/\gamma}]$. En effet, les deux cas extrêmes sont

- $\beta = 0$ signifie que tous les chemins sont optimaux — c'est à dire que toutes les récompenses sont les mêmes. Ceci correspond à $\kappa = K$.
- Si il n'y a qu'un seul chemin où les récompenses sont égales à 1 et que toutes les autres récompenses sont égales à 0 alors, pour tout ϵ, la proportion de nœuds de profondeur d ϵ-optimaux est $1/K^d$ pour $d \leq d_0$ pour une profondeur d_0 pour laquelle $\frac{\gamma^{d_0}}{1-\gamma} \geq \epsilon$ et, pour tout $d > d_0$, la proportion de nœuds ϵ-optimaux reste constante. Puisque $d_0 \geq \log_\gamma(1-\gamma)\epsilon$, la proportion de nœuds de profondeur $d > d_0$ ϵ-optimaux est au plus $[(1-\gamma)\epsilon]^{\frac{\log K}{\log 1/\gamma}}$, c'est-à-dire que $\beta = \frac{\log K}{\log 1/\gamma}$ ce qui est la valeur la plus élevée pour β correspondant à $\kappa = 1$.

De ce résultat, nous constatons que quand $\kappa > 1$, le taux de convergence du regret pour l'algorithme de planification optimiste est $n^{-\frac{\log 1/\gamma}{\log \kappa}}$ en lieu et place de $n^{-\frac{\log 1/\gamma}{\log K}}$ pour l'algorithme de planification uniforme. En examinant la preuve, nous pouvons remarquer que κ joue le rôle du facteur de branchement pour l'arbre $\mathcal{T}_\infty$ des nœuds étendus tout comme K est le facteur de branchement de l'arbre infini $\mathcal{T}$. κ appartenant à l'intervalle $[1, K]$, le taux de décroissance du regret de l'algorithme de planification optimiste est toujours au moins aussi bon que celui de l'algorithme de planification uniforme. La borne pour l'algorithme de planification optimiste est meilleure que celle pour l'algorithme de planification uniforme quand $\kappa < K$ et encore plus quand κ est proche de 1. Notons que l'arbre $\mathcal{T}_\infty$ représente l'ensemble des nœuds i tel que, étant donné les récompenses observées depuis la racine jusqu'au nœud i, nous ne pouvons pas décider si i appartient à un chemin optimal ou pas. Cela représente l'ensemble des nœuds qui doivent être étendus pour définir un chemin optimal. Ainsi les performances de l'algorithme de planification optimiste sont exprimées en fonction du facteur de branchement du sous-arbre $\mathcal{T}_\infty$ composé de tous les nœuds devant être étendu éventuellement.

4.3.3 Comparaison avec l'algorithme A*

Dans le chapitre précédent, nous avons présenté l'algorithme A* avec lequel l'algorithme de planification optimiste partage certains points communs. Bien que l'algorithme A* cherche à minimiser la distance parcourue — autrement dit minimiser un coût — plutôt que de maximiser une somme de récompenses dépréciées comme l'algorithme de planification optimiste, tous deux sélectionnent un nœud à explorer possédant la borne la plus petite pour le premier ou la plus grande pour le second. De plus la structure même de la borne est similaire. En effet dans le cas de l'algorithme A*, la borne est constituée de la distance du nœud de départ au nœud courant et d'une borne inférieure sur le distance restante. Alors que dans le cas de l'algorithme de planification optimiste, la borne est constituée de la somme des récompenses dépréciées du nœud racine au nœud courant et d'une borne supérieure sur la somme des récompenses dépréciées accessibles depuis le nœud courant.

Cependant dans le cadre de la planification optimiste, un nœud final n'est pas obligatoire pour le calcul de la borne supérieure alors que dans le cas l'algorithme A*, la borne inférieure sur la distance restante est calculée à partir du nœud courant jusqu'à l'un des nœuds finaux. L'algorithme de planification optimiste travaille donc dans un espace de chemins infinis contrairement à l'algorithme A*.

Il est à noter que la borne inférieure sur la distance entre un nœud courant et l'ensemble des nœuds finaux est une heuristique alors que dans le cadre

de l'algorithme de planification optimiste, la borne supérieure sur la somme des récompenses dépréciées accessibles depuis le nœud courant provient du fait que les récompenses sont dépréciées dans le temps.

Il est à noter également que l'algorithme A* construit un sous-graphe du graphe original alors que l'algorithme de planification optimiste ne procède pas à des tests de redondance sur les états rencontrés. Cette absence de tests est dû à une volonté de rester indépendant de la dimension de l'espace d'état dans la définition de l'algorithme. Dans la pratique, fusionner les nœuds associés au même état peut améliorer l'efficacité de l'algorithme — en particulier dans le cadre d'un espace d'état discret — puisque les ressources computationnelles sont ainsi regroupées sur un même sous-graphe et non réparties sur plusieurs instances du même sous-arbre.

Comme nous l'avons montré précédemment, l'arbre $\mathcal{T}_\infty$ comprend les nœuds qui doivent être étendu pour savoir si ils appartiennent ou non à un chemin optimal, cet arbre étant celui construit par l'algorithme de planification optimiste. De la même manière, l'algorithme A* explore seulement les nœuds devant l'être pour déterminer si effectivement ils appartiennent ou non au plus court chemin si la borne inférieure sur la distance d'un nœud à l'ensemble des nœuds finaux vérifie l'équation (3.3) à la page 44.

Dans la section suivante, nous allons présenter les différentes expérimentations effectuées sur les algorithmes de planification uniforme et optimiste ainsi que sur les algorithmes UCT modifié et de recherche aléatoire.

4.4 Expérimentations

Dans un premier temps, nous allons définir la version modifiée de l'algorithme UCT ainsi que l'algorithme de recherche aléatoire. Puis dans un second temps, nous allons décrire les résultats des différentes expérimentations sur les cinq problèmes considérés en donnant au préalable leur définition.

4.4.1 Définition des algorithmes UCT modifié et de recherche aléatoire

Les algorithmes UCT et Rollout ont été adaptés au cas présent où les systèmes considérés sont déterministes. Ainsi l'algorithme UCT modifié n'effectuera pas d'appels au modèle génératif lors de la descente en profondeur de l'arbre des possibilités, les nœuds étendus précédemment étant parfaitement connus. De plus il est à noter que comme la planification uniforme et optimiste, l'algorithme UCT modifié appellera le modèle génératif pour chacune des actions de la feuille atteinte lors de la descente contrairement à l'algorithme UCT. Enfin la valeur à maximiser parmi les actions $a \in A$ pour un état $x \in \mathcal{S}_n$ de profondeur d ne sera pas $\hat{Q}(x,a) + 2C_p\sqrt{\frac{\ln n_x}{n_{x,a}}}$ mais $u_x + \frac{\gamma^d}{1-\gamma}\sqrt{\frac{\ln n_x}{n_{x,a}}}$.

Algorithme 4.3 Recherche Aléatoire

$H \leftarrow 1$
$n \leftarrow 0$
$\tilde{Q}(x_0, a) \leftarrow 0, \forall a \in A$
while Une condition d'arrêt n'est pas vraie **do**
  $v \leftarrow 0$
  for $t \leftarrow 0$ **to** $H-1$ **do**
    $a_t \sim \mathcal{U}_A$
    $v \leftarrow v + \gamma^t r(x_t, a_t)$
    $n \leftarrow n + 1$
  end for
  if $\tilde{Q}(x_0, a_0) < v$ **then**
    $\tilde{Q}(x_0, a_0) \leftarrow v$
  end if
  $H \leftarrow \left\lfloor 1 + \frac{\log n}{\log(1-\gamma)} \right\rfloor$
end while
return $\arg\max_{a \in A} \tilde{Q}(x_0, a)$

Le principe de l'algorithme de recherche aléatoire — décrit par l'algorithme 4.3 — est de construire des trajectoires aléatoires dont la longueur augmentera au cours du temps. Cette longueur sera la partie entière de $1 + \log n / \log(1-\gamma)$ où n est le nombre d'appels au modèle génératif déjà

effectués. Faire évoluer la longueur permet de prendre en compte le fait que le nombre d'appels au modèle génératif est inconnu et ainsi d'obtenir un comportement anytime. La longueur d'une trajectoire correspond au nombre d'appels au modèle génératif nécessaires pour construire la trajectoire. Chaque action le long d'une trajectoire sera issue d'un tirage uniforme discret sur A tel que $a_t \sim \mathcal{U}_A$ avec $t \in \{0, \dots, H-1\}$. Lorsque les ressources computationnelles sont épuisées, la première action de la trajectoire dont la somme des récompenses dépréciées est la plus élevée est retournée.

4.4.2 Le problème de la balle

Le problème de la balle a pour objectif de faire atteindre un point à une balle se mouvant le long d'un axe en contrôlant son accélération — voir la figure 4.3. Les dynamiques du système sont $(p_{t+1}, v_{t+1})' = (p_t, v_t)' + (v_t, a_t)'\Delta t$ où p_t est la position de la balle sur l'axe à l'instant t, $v_t \in [-2, 2]$ est la vitesse de la balle à l'instant t, $a_t \in A$ est l'accélération de la balle avec $A = \{-1, 1\}$ et $\Delta t = 0.1$ est le pas de temps.

L'état du système à l'instant t est représenté par le couple (p_t, v_t). La fonction récompense pour un état $x_t = (p_t, v_t)$ et une action a_t est définie par $r(x_t, a_t) = \max(1 - p_{t+1}^2, 0)$. La facteur de dépréciation γ est de 0.9.

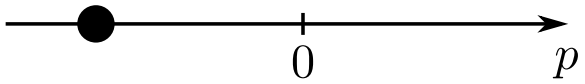


FIGURE 4.3 – Le problème de la balle.

La première expérimentation effectuée sur le problème de la balle concerne le regret défini par l'équation (4.1). Les graphiques 4.4, 4.5 et 4.6 représentent le regret moyen calculé sur 1000 états initiaux tirés uniformément sur le domaine $[-1, 1] \times [-1, 1]$ pour trois facteurs de branchement différents $K = 2$, $K = 3$ et $K = 5$ et dont les actions sont respectivement $\{-1, 1\}$, $\{-1, 0, -1\}$ et $\{-1, -0.5, 0, 0.5, 1\}$. Les différents algorithmes de planification se voient attribués un nombre d'appels au modèle génératif n correspondant au nombre d'appels au modèle génératif nécessaire pour construire un arbre des possibilités uniforme pour une profondeur d et un facteur de branchement K donnés tels que $n = \sum_{i=0}^{d} K^{i+1}$.

Le premier graphique 4.4 représente le regret moyen pour le facteur de branchement $K = 2$ et une profondeur $d \in [0, 21]$. Nous pouvons observer que l'algorithme de planification optimiste — courbe rouge — ne fait pas pire que l'algorithme de planification uniforme — courbe turquoise — comme suggéré par les bornes supérieures sur leur regret. De plus, nous pouvons en déduire que le facteur de branchement κ de l'arbre $\mathcal{T}_\infty$ est très proche du facteur de branchement K de l'arbre $\mathcal{T}$ impliquant que la borne supérieure sur le regret de l'algorithme de planification est proche de celle sur le regret de l'algorithme de planification uniforme. Nous pouvons voir aussi que le

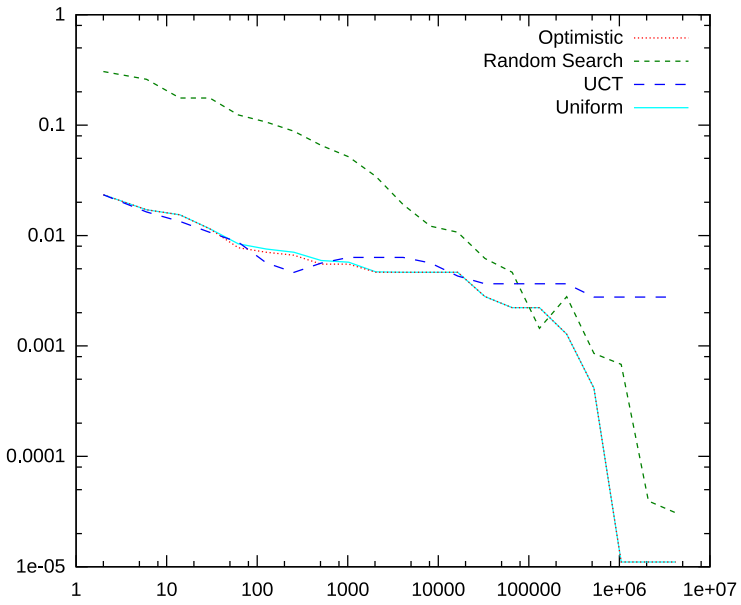


FIGURE 4.4 – Regret moyen pour le problème de la balle avec $K = 2$.

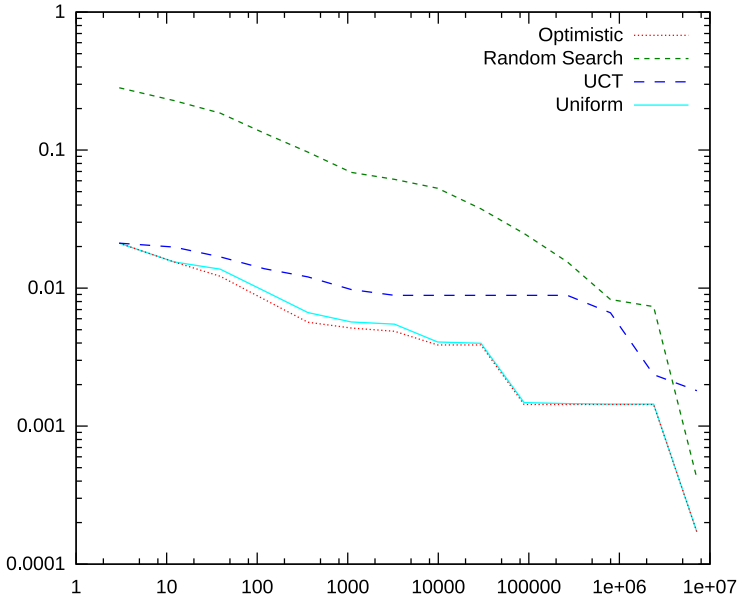


FIGURE 4.5 – Regret moyen pour le problème de la balle avec $K = 3$.

regret de l'algorithme UCT modifié — courbe bleue — ne tend pas aussi vite vers 0 que les trois autres algorithmes. Enfin l'algorithme de recherche aléatoire — courbe verte — possède un regret bien plus élevé au départ mais

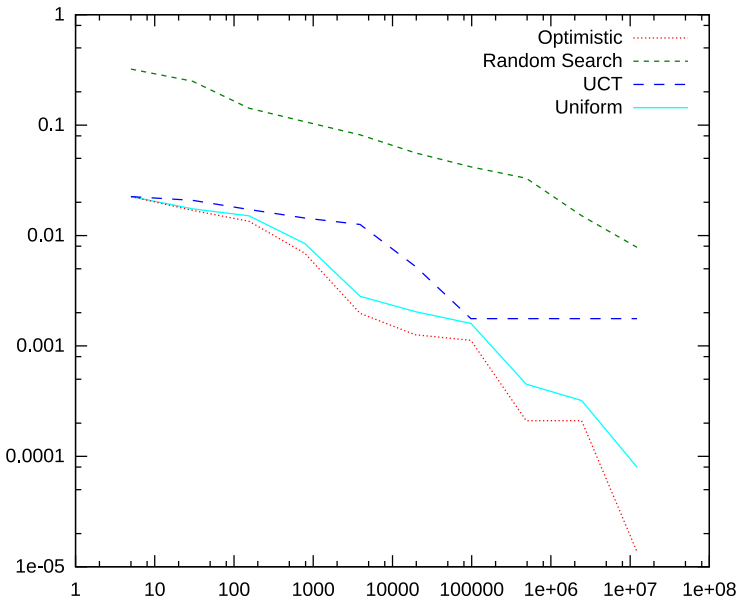


FIGURE 4.6 – Regret moyen pour le problème de la balle avec $K = 5$.

tend vers 0 au fur et à mesure que des ressources computationnelles lui sont accordées.

Le deuxième graphique 4.5 représente le regret moyen mais cette fois ci pour le facteur de branchement $K = 3$ et une profondeur $d \in [0, 13]$. De nouveau nous pouvons observer la même tendance que précédemment où le regret de l'algorithme de planification optimiste n'est pas pire que celui de l'algorithme de planification uniforme. Dans l'ensemble, les performances des quatre algorithmes sont dégradées de part l'augmentation du facteur de branchement.

Enfin le troisième graphique 4.6 représente le regret moyen pour le facteur de branchement $K = 5$ et une profondeur $d \in [0, 9]$. Nous pouvons observer une différence visible entre le regret de l'algorithme de planification optimiste et le regret de l'algorithme de planification uniforme. Cette différence peut s'expliquer par un facteur de branchement κ de l'arbre $\mathcal{T}_\infty$ plus faible que le facteur de branchement K qui privilégie l'algorithme de planification optimiste dont la borne supérieure sur le regret est dépendante de κ comme nous l'avons montré précédemment.

La deuxième expérimentation effectuée sur le problème de la balle consiste à additionner les récompenses obtenues à chaque prise de décision et ce pour 100 pas de temps, les décisions prises étant celles retournées par les algorithmes de planification. La moyenne de la somme de récompenses sur 1000 états initiaux tirés uniformément sur le domaine $[-1, 1] \times [-1, 1]$ est calculée pour différents nombres maximums d'appels n en fonction de la

profondeur d et du facteur de branchement K.

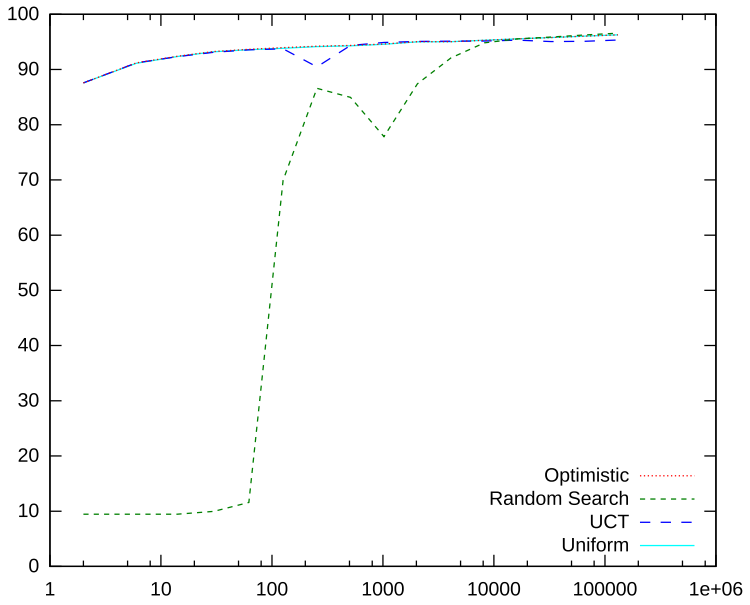


FIGURE 4.7 – Somme des récompenses pour le problème de la balle avec $K = 2$.

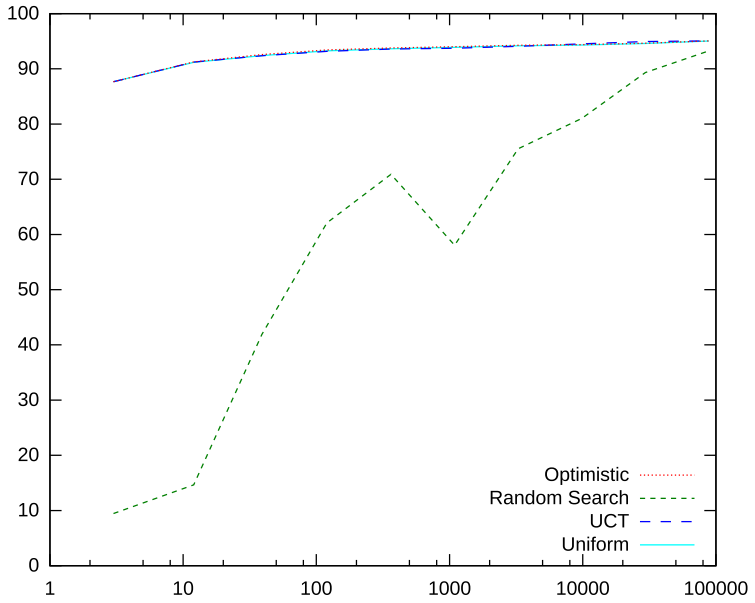


FIGURE 4.8 – Somme des récompenses pour le problème de la balle avec $K = 3$.

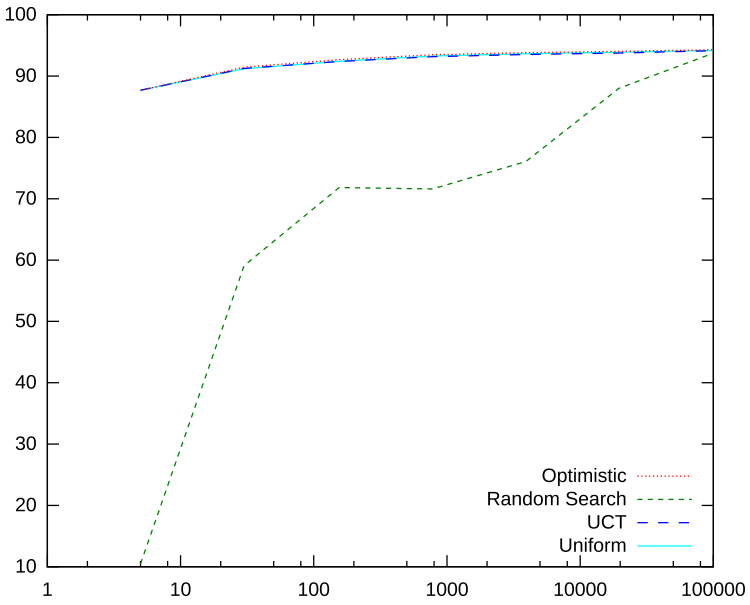


FIGURE 4.9 – Somme des récompenses pour le problème de la balle avec $K = 5$.

Le premier graphique 4.7 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 2$ et pour une profondeur $d \in [0, 15]$. Nous pouvons observer que les performances des algorithmes de planification optimiste et uniforme ainsi que de l'algorithme UCT modifié sont identiques bien que ce dernier soit moins stable. L'algorithme de recherche aléatoire qui est nettement en retrait reprend le dessus à la fin. Le deuxième graphique 4.8 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 3$ et pour une profondeur $d \in [0, 9]$. Le troisième graphique 4.9 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 5$ et pour une profondeur $d \in [0, 7]$. Comme sur le premier graphique, les performances pour les algorithmes de planification optimiste et uniforme et UCT modifié sont identiques.

Comme nous pouvons le voir, les résultats sont très proches d'un algorithme à l'autre et ce pour chacun des facteurs de branchement. Ce comportement est dû en majorité à la fonction récompense qui est très directive dans sa définition. De fait, l'algorithme de recherche aléatoire obtient logiquement des performances plus faible que les algorithmes prenant en compte la récompense dans leur méthode d'exploration et d'exploitation.

4.4.3 Le problème du cart-pole

Le problème du cart-pole a pour objectif de mettre en équilibre vertical un mat accroché par une de ses extrémités à un chariot se déplaçant le long

d'un axe sur lequel une force peut être exercé horizontalement — voir la figure 4.10. Les dynamiques du système — provenant de [Coulom, 2002] — sont décrites par le système d'équations suivant :

$$\begin{pmatrix} \frac{4}{3}l & -\cos\theta \\ lm_m\cos\theta & -(m_c+m_m) \end{pmatrix} \begin{pmatrix} \ddot{\theta} \\ \ddot{p} \end{pmatrix} = \begin{pmatrix} g\sin\theta - \frac{\mu_m\dot{\theta}}{lm_m} \\ lm_m\dot{\theta}^2\sin\theta - a + \mu_c\text{sign}(\dot{p}) \end{pmatrix}$$

où $a \in A$ est la force exercée sur le chariot avec $A = \{-10, 10\}$, $p \in [-2.4, 2.4]$ est la position du chariot sur l'axe, $\theta \in [0, 2\pi]$ est la position angulaire du mat, $l = 0.5$ est la demi-longueur du mat, $m_c = 1$ est la masse du chariot, $m_m = 0.1$ est la masse du mat, μ_c est le coefficient de frottement avec l'axe du chariot, μ_m est le coefficient de frottement du mat avec le chariot et $\text{sign}(\dot{p})$ est une fonction de $\dot{p}$ qui retourne -1 si $\dot{p}$ est négatif, 1 sinon.

L'état du système à l'instant t est représenté par le quadruplet $(p_t, \theta_t, \dot{p}_t, \dot{\theta}_t)$. La fonction récompense est différente de celle définie par [Coulom, 2002] pour que les récompenses soient entre 0 et 1 et est définie pour un état $x_t = (p_t, \theta_t, \dot{p}_t, \dot{\theta}_t)$ et une action a_t par l'équation $r(x_t, a_t) = (1+\cos\theta_{t+1})/2$. Cependant si $|p_{t+1}| > L$ avec $L = 2.4$ la demi-longueur de la piste, la récompense est égale à 0. Le facteur de dépréciation γ est de 0.95 et le pas de temps est de 0.1.

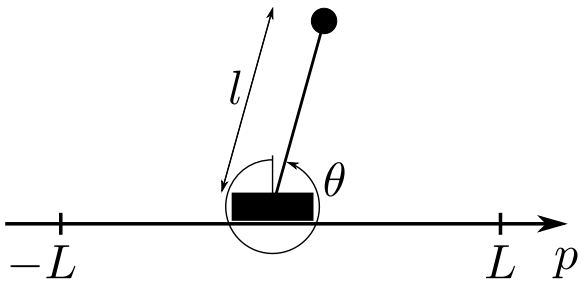


FIGURE 4.10 – Le problème du cart-pole.

L'expérimentation sur le problème du cart-pole consiste à calculer la moyenne sur 1000 états initiaux pour différents facteurs de branchement K et pour différentes quantités de ressources computationnelles n de la somme des récompenses sur 100 pas de temps. Les états initiaux ont été tiré uniformément dans le domaine $[-2, 2] \times [-5, 5] \times [1, 5.28] \times [-1, 1]$. Les facteurs de branchements utilisés sont $K = 2$, $K = 3$ et $K = 5$ avec respectivement comme ensemble d'actions $\{-10, 10\}$, $\{-10, 0, 10\}$ et $\{-10, -5, 0, 5, 10\}$.

Le premier graphique 4.11 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 2$ et une profondeur $d \in [0, 15]$. Nous observons qu'initialement les algorithmes de planification optimiste et uniforme ainsi que l'algorithme UCT modifié possèdent la même moyenne sur la somme des récompenses ce qui est logique car lorsque seul l'état initial est étendu, aucune feuille ne doit être sélectionné pour être étendu. Ainsi ils

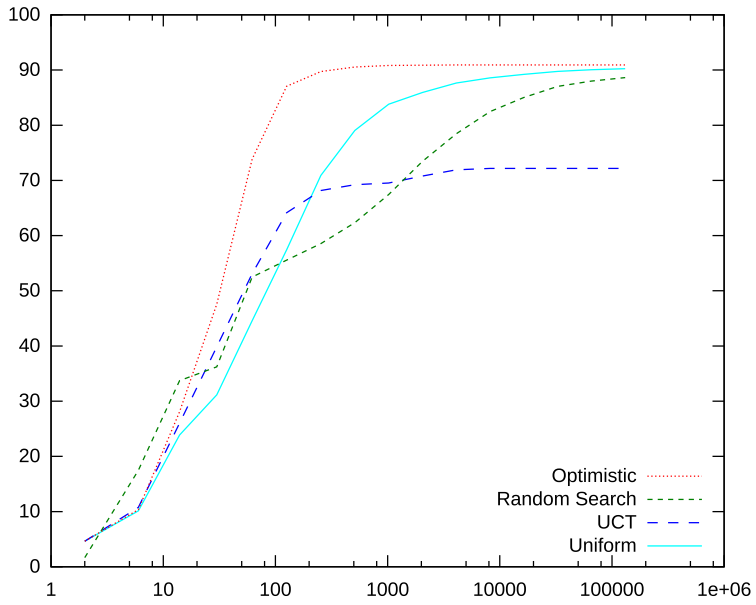


FIGURE 4.11 – Somme des récompenses pour le problème de cart-pole avec $K = 2$.

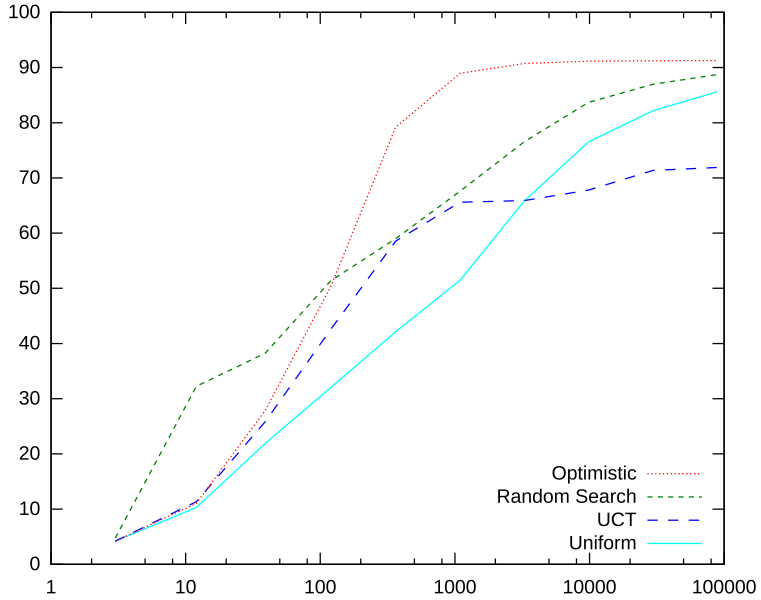


FIGURE 4.12 – Somme des récompenses pour le problème de cart-pole avec $K = 3$.

ont tous trois le même comportement. Après ce point, l'ordre d'exploration

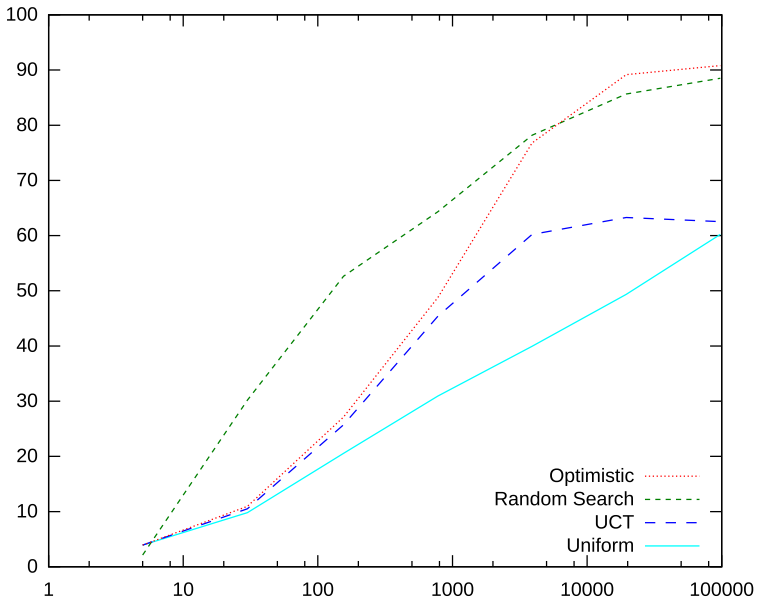


FIGURE 4.13 – Somme des récompenses pour le problème de cart-pole avec $K = 5$.

de l'arbre $\mathcal{T}$ est différent pour chacun de ces trois algorithmes ce qui est reflété dans leur résultat respectif. Ainsi lorsque $n = 14$ appels au modèle génératif sont alloués, l'algorithme de planification optimiste fait mieux que l'algorithme UCT modifié et pour $n = 30$, il fait mieux que l'algorithme de recherche aléatoire. Nous observons aussi qu'avec $n = 254$ appels au modèle génératif, l'algorithme de planification uniforme fait mieux que l'algorithme UCT modifié. Il est à noter qu'à partir de $n = 8190$, les résultats obtenus avec l'algorithme de planification optimiste ne changent plus. Il en est de même pour l'algorithme UCT modifié à partir de $n = 16382$.

Le deuxième graphique 4.12 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 3$ et une profondeur $d \in [0, 9]$. L'augmentation du facteur de branchement fait diminuer les performances des quatre algorithmes bien que l'algorithme de recherche aléatoire soit moins affecté que les autres. Nous pouvons ainsi remarquer que l'algorithme de recherche aléatoire domine l'algorithme de planification uniforme ainsi que l'algorithme UCT modifié. Cependant l'algorithme de planification optimiste dépasse l'algorithme de recherche aléatoire quand $n \geq 363$.

Le troisième graphique 4.13 représente la moyenne de la somme des récompense pour un facteur de branchement $K = 5$ et une profondeur $d \in [0, 7]$. Nous pouvons observer une baisse des performances bien que l'algorithme de recherche aléatoire conserve des performances comparables car étant moins sensible au facteur de branchement. L'algorithme de plani-

fication uniforme est presque dominé par tous les autres algorithmes du fait de sa forte dépendance au facteur de branchement en terme de profondeur d'exploration pour une même quantité de ressources computationnelles.

Contrairement au problème de la balle, la fonction récompense du problème du cart-pole est moins directive et possède des maxima locaux pouvant conduire à un comportement sous-optimal. En effet, un mouvement de balancier doit être appliqué au mât pour augmenter sa vitesse angulaire et le mettre en équilibre vertical. Or, pendant ce mouvement de balancier, les récompenses obtenues baissent, ne favorisant donc pas l'exploitation des séquences associées dans l'arbre des possibilités. Il faut pour cela explorer des parties à première vue sous-optimales pour découvrir des séquences d'actions dont la somme des récompenses dépréciées est plus élevée par la suite. La répartition entre exploration et exploitation est donc ici importante.

4.4.4 Le problème de l'acrobot

L'acrobot est un robot composé de deux segments : les jambes et le corps — voir la figure 4.14. Le corps est monté sur un axe fixe sur une de ses extrémités. L'autre extrémité est une articulation reliant le corps aux jambes sur laquelle un couple peut être appliqué. L'objectif est de mettre en équilibre vertical les deux segments. Les dynamiques du système — provenant de [Coulom, 2002] — sont décrites par le système d'équations suivant :

$$\begin{pmatrix} a_1 & a_3 \\ a_3 & a_2 \end{pmatrix} \begin{pmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

avec

$$\begin{cases} a_1 &= (\frac{4}{3}m_1 + 4m_2)l_1^2, \\ a_2 &= \frac{4}{3}m_2 l_2^2, \\ a_3 &= 2m_2 l_1 l_2 \cos(\theta_1 - \theta_2), \\ b_1 &= 2m_2 l_2 l_1 \dot{\theta}_2^2 \sin(\theta_2 - \theta_1) + (m_1 + 2m_2) l_1 g \sin\theta_1 - \mu_1 \dot{\theta}_1 - a, \\ b_2 &= 2m_2 l_1 l_2 \dot{\theta}_1^2 \sin(\theta_1 - \theta_2) + m_2 l_2 g \sin\theta_2 - \mu_2 \dot{\theta}_2 + a \end{cases}$$

où $a \in A$ est le couple exercé sur l'articulation entre le corps et les jambes de l'acrobot avec $A = \{-2, 2\}$, $\theta_1 \in [0, 2\pi]$ est la position angulaire du corps, $\theta_2 \in [0, 2\pi]$ est la position angulaire des jambes, $m_1 = 1$ est la masse du corps, $m_2 = 1$ est la masse des jambes, $l_1 = 0.5$ est la demi-longueur du corps, $l_2 = 0.5$ est la demi-longueur des jambes, $\mu_1 = 0.05$ est le coefficient de friction entre le corps et l'axe fixe et $\mu_2 = 0.05$ est le coefficient de friction de l'articulation corps-jambes.

L'état du système à l'instant t est représenté par le quadruplet $(\theta_{1,t}, \theta_{2,t}, \dot{\theta}_{1,t}, \dot{\theta}_{2,t})$. La fonction récompense est différente de celle définie par [Coulom, 2002] pour que les récompenses soient entre 0 et 1 et pour qu'elle soit aussi plus directrice. Elle est définie pour un état $x_t = (\theta_{1,t}, \theta_{2,t}, \dot{\theta}_{1,t}, \dot{\theta}_{2,t})$ et une

action a_t par l'équation suivante :

$$\begin{aligned} r(x_t, a_t) &= 1 - \big\{(l_1 \sin\theta_{1,t+1} + l_2 \sin\theta_{2,t+1})^2 \\ &\quad + (l_1 \cos\theta_{1,t+1} + l_2 \cos\theta_{2,t+1} - (l_1 + l_2))^2\big\}^{\frac{1}{2}} / (2(l_1 + l_2)). \end{aligned}$$

Le facteur de dépréciation γ est de 0.95 et le pas de temps de 0.1.

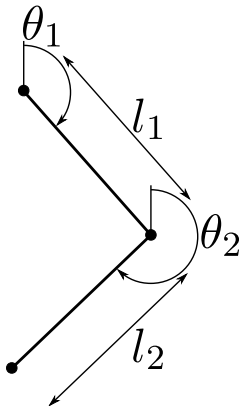


FIGURE 4.14 – Le problème de l'acrobot.

L'expérimentation sur le problème de l'acrobot consiste à calculer la moyenne sur 1000 états initiaux et pour différents facteurs de branchement K et nombres n d'appels au modèle génératif de la somme des récompenses sur 100 pas de temps. Les états initiaux sont tirés uniformément sur le domaine $[1, 5.28] \times [-1, 1] \times [1, 5.28] \times [-1, 1]$. Les facteurs de branchement utilisés sont $K = 2$, $K = 3$ et $K = 5$ avec respectivement comme ensemble d'actions $\{-2, 2\}$, $\{-2, 0, 2\}$ et $\{-2, -1, 0, 1, 2\}$.

Le premier graphique 4.15 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 2$ et une profondeur $d \in [0, 15]$. Comparé aux trois autres algorithmes, l'algorithme de recherche aléatoire est en retrait. L'algorithme de planification optimiste se détache des algorithmes de planification uniforme et UCT modifié pour $n \geq 126$. Contrairement au problème précédent, l'algorithme de planification uniforme à l'avantage sur l'algorithme UCT modifié qui se fait dépasser pour $n \geq 32766$ par l'algorithme de recherche aléatoire.

Le deuxième graphique 4.16 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 3$ et une profondeur $d \in [0, 9]$. Parmi les quatre algorithmes, seul l'algorithme de recherche aléatoire ne voit pas ses performances baisser significativement par l'augmentation du facteur de branchement. Comme précédemment, l'algorithme de planification optimiste fait mieux que l'algorithme de planification uniforme, lui même faisant mieux que l'algorithme UCT modifié.

Le troisième graphique 4.17 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 5$ et une profondeur

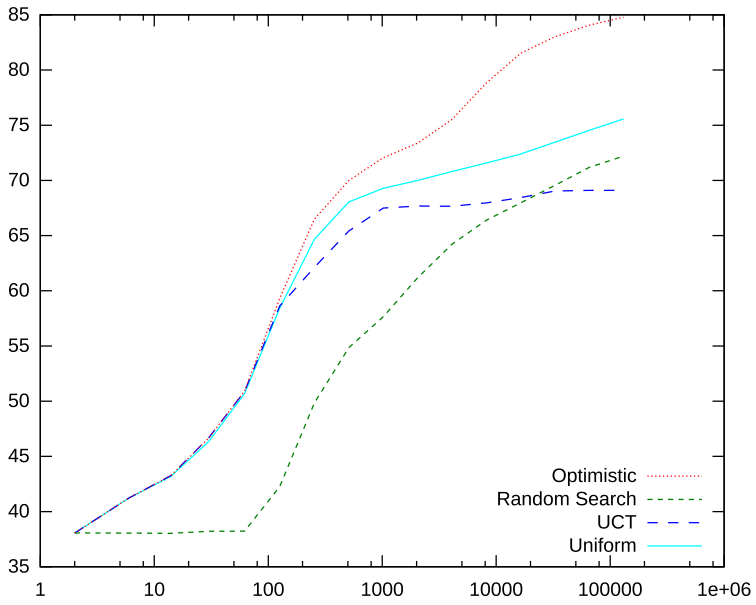


FIGURE 4.15 – Somme des récompenses pour le problème de l'acrobot avec $K = 2$.

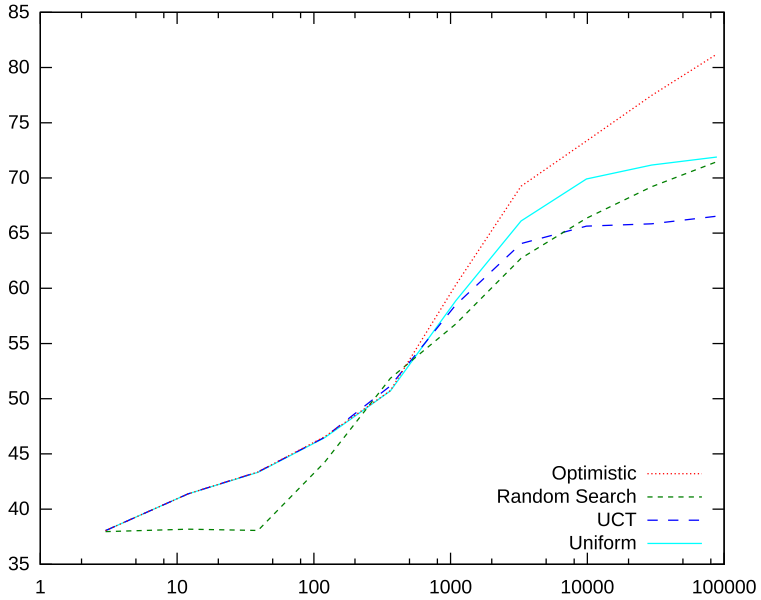


FIGURE 4.16 – Somme des récompenses pour le problème de l'acrobot avec $K = 3$.

$d \in [0, 7]$. Ce graphique confirme la tendance à la baisse des performances et

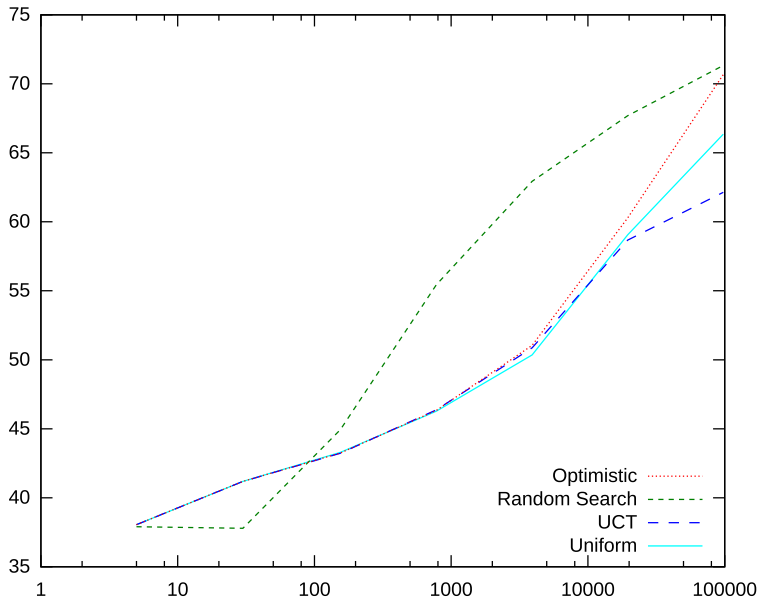


FIGURE 4.17 – Somme des récompenses pour le problème de l'acrobot avec $K = 5$.

le maintient de l'algorithme de recherche aléatoire en dépit de l'augmentation du facteur de branchement.

Le problème de l'acrobot est semblable au problème du cart-pole dans la définition de sa fonction récompense et dans les conséquences qu'elle entraine. Pour mettre en équilibre l'acrobot et suivant la définition de la fonction récompense, un mouvement entrainant une alternance de récompenses élevées et faibles est requis. Cette alternance est renforcée par la définition de la fonction récompense impliquant la position angulaire du corps et des jambes. De plus, le maintient de l'acrobot dans une position d'équilibre — et donc proche de la récompense maximale — est plus compliqué par nature que dans le cas du cart-pole. Nous pouvons ainsi observer une plus forte dégradation des performances lorsque le facteur de branchement augmente compliquant ainsi l'exploration de l'arbre des possibilités.

4.4.5 Le problème du mountain car

Le problème du mountain car a pour objectif de faire monter une voiture en haut d'une colline en contrôlant son accélération — voir la figure 4.18. Les dynamiques du système — provenant de [Ernst et al., 2005] — sont décrites par les deux équations différentielles suivantes :

$$\begin{aligned} \dot{p} &= v \\ \dot{v} &= \frac{a}{m(1+\mathrm{Hill}'(p)^2)} - \frac{g\mathrm{Hill}'(p)}{1+\mathrm{Hill}'(p)^2} - \frac{v^2\mathrm{Hill}'(p)\mathrm{Hill}''(p)}{1+\mathrm{Hill}'(p)^2} \end{aligned}$$

où $a \in A$ est l'accélération de la voiture avec $A = \{-4, 4\}$, $v \in [-3, 3]$ est la vitesse de la voiture, $p \in [-1, 1]$ est la position de la voiture, $m = 1$ est la masse de la voiture, $g = 9.81$ est la pesanteur et $\text{Hill}(p)$ est une fonction de p représentant la colline et dont la définition est la suivante :

$$\text{Hill}(p) = \begin{cases} p^2 + p & \text{si } p < 0 \\ \frac{p}{\sqrt{1+5p^2}} & \text{si } p \geq 0 \end{cases} .$$

L'état du système à l'instant t est représenté par le couple (p_t, v_t). La fonction récompense est différente de celle définie par [Ernst et al., 2005] pour que les récompenses soient entre 0 et 1 et pour qu'elle soit aussi plus directrice. Elle est définie pour un état $x_t = (p_t, v_t)$ et une action a_t par l'équation $r(x_t, a_t) = \frac{p_{t+1}+1}{2}$. Cependant si $p_{t+1} < -1$ ou que $|v_{t+1}| > 3$, la récompense est égale à 0. Le facteur de dépréciation γ est de 0.95 et le pas de temps est de 0.1. Les dynamiques du système sont intégrées en utilisant la méthode d'Euler avec un pas de temps d'intégration de 0.0001.

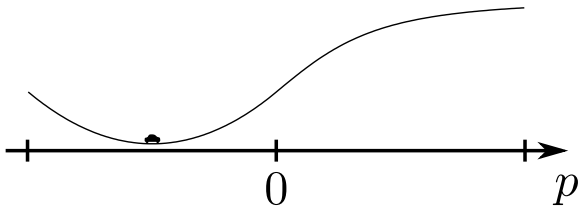


FIGURE 4.18 – Le problème du mountain car.

L'expérimentation sur le problème du mountain car consiste à calculer la moyenne sur 1000 états initiaux et pour différents facteurs de branchements K et nombres n d'appels au modèle génératif de la somme des récompense sur 100 pas de temps. Les états initiaux sont tirés uniformément sur le domaine $[-0.75, 0.75] \times [-1, 1]$. Les facteurs de branchement utilisés sont $K = 2$, $K = 3$ et $K = 5$ avec respectivement comme ensemble d'actions $\{-4, 4\}$, $\{-4, 0, 4\}$ et $\{-4, -2, 0, 2, 4\}$.

Le premier graphique 4.19 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 2$ et une profondeur $d \in [0, 15]$. L'algorithme de planification optimiste — comme l'algorithme de planification uniforme — produit les mêmes résultats jusqu'à $n = 2046$ pour après voir ses performances augmenter et dépasser celles des algorithmes UCT modifié et de recherche aléatoire. Cette stagnation initiale vient de la répartition des récompenses dans l'arbre $\mathcal{T}$. Le problème du mountain car induit un mouvement de balancier pour augmenter la vitesse de la voiture et lui permettre de monter la pente. Or suivant la définition de la fonction récompense lorsque la voiture s'éloigne de l'objectif, la récompense obtenue diminue ce qui a pour effet de produire des séquences d'actions optimales alternant des récompenses élevées et faibles. Ainsi le comportement de l'algorithme de planification optimiste face à de telles séquences d'actions est

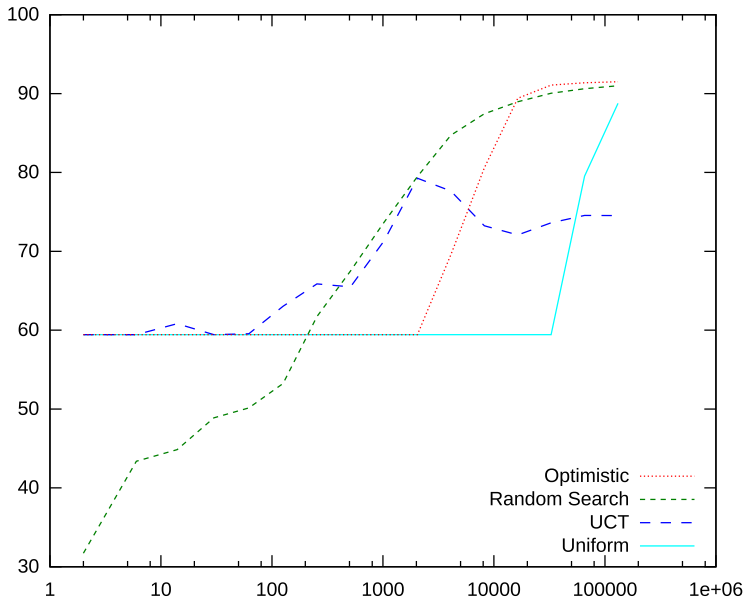


FIGURE 4.19 – Somme des récompenses pour le problème du mountain car avec $K = 2$.

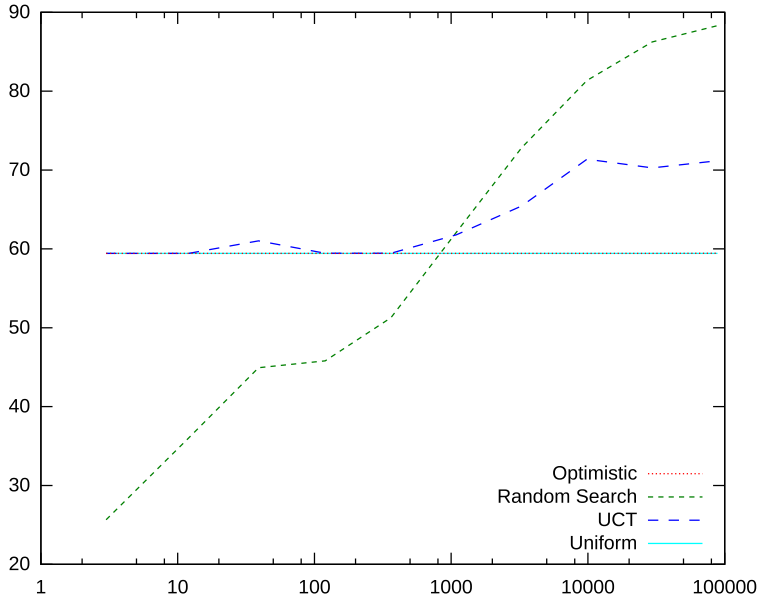


FIGURE 4.20 – Somme des récompenses pour le problème du mountain car avec $K = 3$.

de privilégier celles menant le plus rapidement aux récompenses élevées et

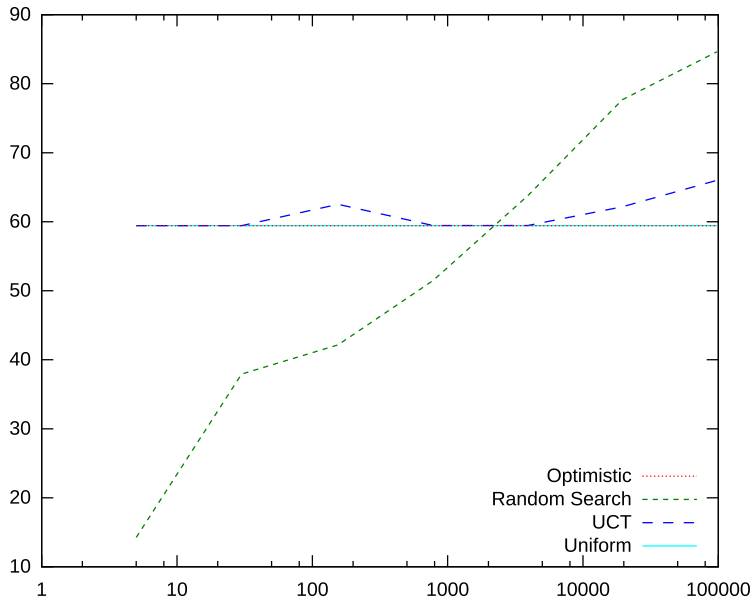


FIGURE 4.21 – Somme des récompenses pour le problème du mountain car avec $K = 5$.

faisant l'impasse sur le mouvement de balancier. La tendance est inversée lorsque l'exploration de l'algorithme de planification optimiste le mène à découvrir des séquences d'actions plus optimales grâce à un nombre suffisant d'appels au modèle génératif. Ce phénomène est présent aussi dans le problème du cart-pole mais les alternances de récompenses élevées et faibles étant plus rapide, l'algorithme de planification optimiste est moins impacté dans son comportement.

Le deuxième graphique 4.20 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 3$ et une profondeur $d \in [0, 9]$. La même situation se produit ici mais le facteur de branchement augmentant, le nombre maximum d'appels au modèle génératif attribué à l'algorithme de planification optimiste — $n = 29523$ — n'est pas suffisant pour que son exploration le mène à de meilleures performances. Comme pour les problèmes précédent, l'augmentation du facteur de branchement n'affecte que très peu les résultats de l'algorithme de recherche aléatoire.

Le troisième graphique 4.21 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 5$ et une profondeur $d \in [0, 7]$. L'augmentation du facteur de branchement n'a pas changé les résultats des algorithmes de planification optimiste et uniforme qui sont identiques à ceux présents sur le graphique précédent. Nous pouvons observer une diminution des performances de l'algorithme UCT modifié dû à l'augmentation du facteur de branchement.

Le problème du mountain car est — tout comme le problème de l'acrobot — semblable au problème du cart-pole. Cependant, et contrairement au problème du cart-pole, les récompenses obtenues pendant le mouvement de balancier ne sont pas symétrique en amplitude ce qui a pour effet d'allonger la période pendant laquelle les récompenses baissent compliquant un peu plus encore l'exploration de l'arbre des possibilités. Cette plus longue période explique la stagnation des algorithmes de planification uniforme et optimiste ceux-ci étant piégés par des maxima locaux.

4.4.6 Le problème du double cart-pole

Le problème du double cart-pole se constitue de deux cart-poles reliés par un ressort et s'influençant l'un l'autre dans leur mouvement horizontal — voir la figure 4.22. Le ressort les reliant est caractérisé par sa raideur k, sa taille au repos Δ, sa taille minimale avant déformation $\Delta_{\min}$ et sa taille maximale avant déformation $\Delta_{\max}$. Les dynamiques des cart-poles sont les mêmes que précédemment.

L'état du système à l'instance t est représenté par l'octuplet $(p_{1,t}, \theta_{1,t}, \dot{p}_{1,t}, \dot{\theta}_{1,t}, p_{2,t}, \theta_{2,t}, \dot{p}_{2,t}, \dot{\theta}_{2,t})$ qui est la combinaison de deux quadruplets issus de deux cart-poles. L'action a présente dans la définition des dynamiques d'un cart-pole sera remplacée par $a_1 - k(\Delta - |p_2 - p_1|)$ pour le premier cart-pole avec $a_1 \in [-10, 10]$ la force lui étant exercée et par $a_2 + k(\Delta - |p_2 - p_1|)$ pour le deuxième cart-pole avec $a_2 \in [-10, 10]$ la force lui étant exercée. La fonction récompense est définie pour un état $x_t = (p_{1,t}, \theta_{1,t}, \dot{p}_{1,t}, \dot{\theta}_{1,t}, p_{2,t}, \theta_{2,t}, \dot{p}_{2,t}, \dot{\theta}_{2,t})$ et une action $a_t = (a_{1,t}, a_{2,t})$ par l'équation suivante :

$$r(x_a, a_t) = \begin{cases} 0 & \text{si } |p_{1,t+1}| > 2.4, \\ 0 & \text{si } |p_{2,t+1}| > 2.4, \\ 0 & \text{si } p_{2,t+1} \leq p_{1,t+1}, \\ 0 & \text{si } |p_{2,t+1} - p_{1,t+1}| < \Delta_{\min}, \\ 0 & \text{si } |p_{2,t+1} - p_{1,t+1}| > \Delta_{\max}, \\ (2 + \cos\theta_{1,t+1} + \cos\theta_{2,t+1})/4 & \text{sinon.} \end{cases}$$

Le facteur de dépréciation γ est de 0.95 et le pas de temps de 0.1.

L'expérimentation sur le problème du double cart-pole consiste à calculer la moyenne sur 1000 états initiaux et pour différents facteurs de branchement K et nombres n d'appels au modèle génératif de la somme des récompenses sur 100 pas de temps. Les états initiaux sont tirés uniformément sur le domaine $[-2, 2] \times [-1, 1] \times [1, 5.28] \times [-1, 1] \times [-2, 2] \times [-1, 1] \times [1, 5.28] \times [-1, 1]$. Nous veillerons cependant à ce que pour chaque état initiaux $x = (p_1, \theta_1, \dot{p}_1, \dot{\theta}_1, p_2, \theta_2, \dot{p}_2, \dot{\theta}_2)$, $p_2 \geq p_1$, $|p_2 - p_1| \geq \Delta_{\min}$ et $|p_2 - p_1| \leq \Delta_{\max}$. Dans le cas contraire p_2 sera initialisé à $p_1 + \Delta_{\min} + 0.01$. De plus si $\dot{p}_1 > 0$ et $\dot{p}_2 < 0$, $\dot{p}_2$ sera initialisé à $\dot{p}_1$. Ces restrictions

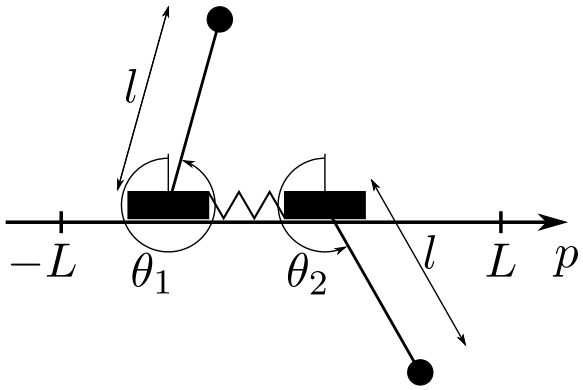


FIGURE 4.22 – Le problème du double cart-pole.

sont nécessaires pour ne pas obtenir des états qui sont terminaux peu importe la première prise de décision. Les facteurs de branchement utilisés sont $K = 4$, $K = 9$ et $K = 25$ avec respectivement comme ensemble d'actions $\{(a_1, a_2)|a_1, a_2 \in \{-10, 10\}\}$, $\{(a_1, a_2)|a_1, a_2 \in \{-10, 0, 10\}\}$ et $\{(a_1, a_2)|a_1, a_2 \in \{-10, -5, 0, 5, 10\}\}$.

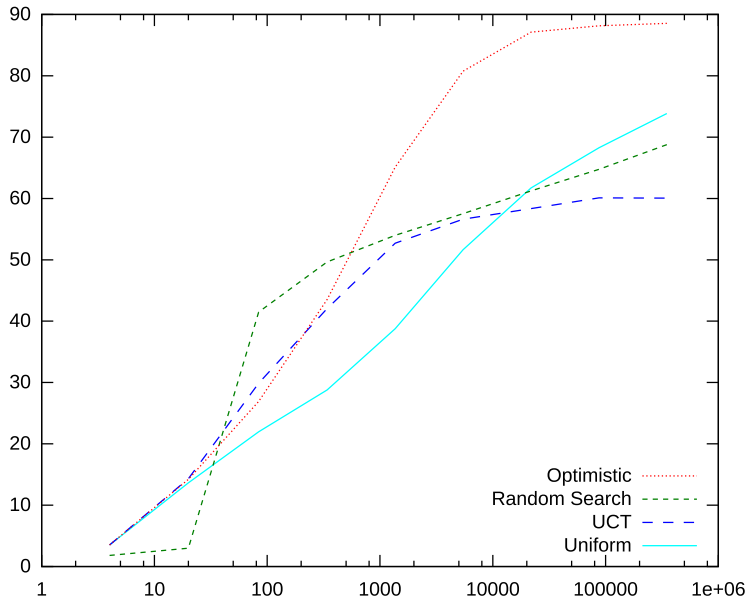


FIGURE 4.23 – Somme des récompenses pour le problème du double cart-pole avec $K = 4$.

Le premier graphique 4.23 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 4$ et une profondeur $d \in [0, 8]$. Dans un premier temps l'algorithme UCT modifié prend l'avantage sur l'al-

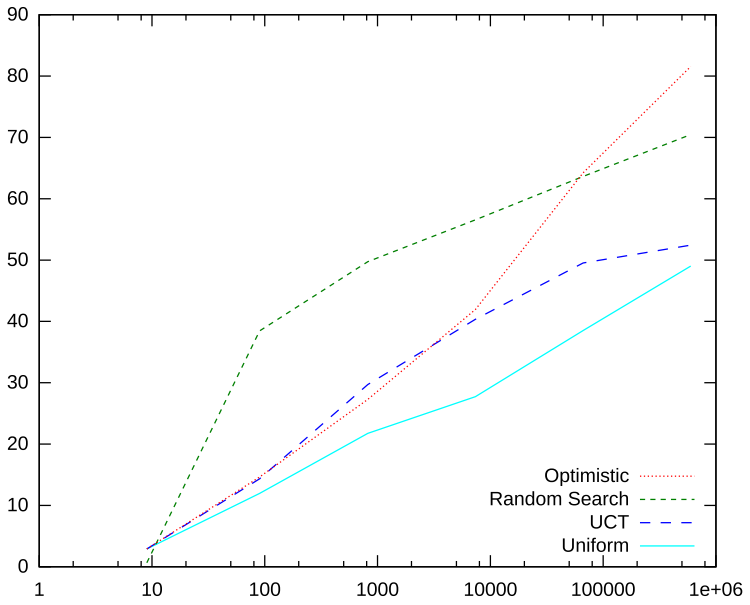


FIGURE 4.24 – Somme des récompenses pour le problème du double cart-pole avec $K = 9$.

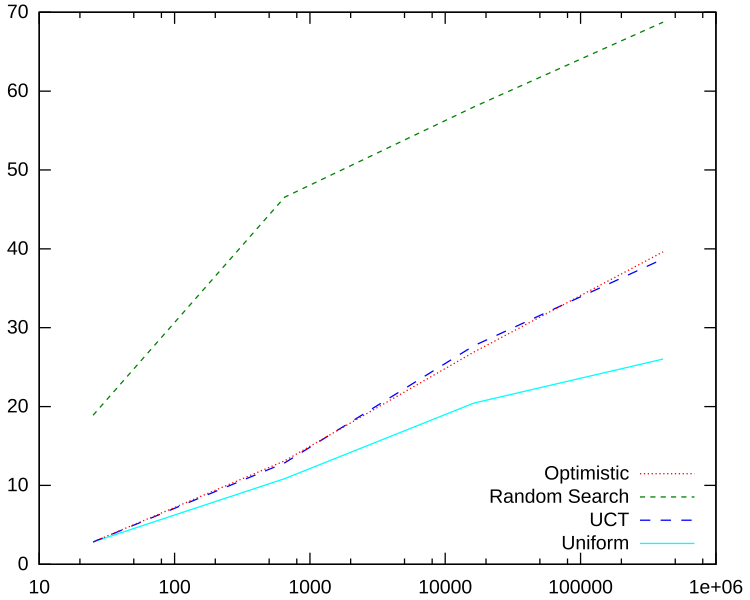


FIGURE 4.25 – Somme des récompenses pour le problème du double cart-pole avec $K = 25$.

gorithme de planification uniforme et optimiste. Puis pour $n \geq 340$, l'algo-

rithme de planification optimiste dépasse l'algorithme UCT modifié et pour $n \geq 1364$, il dépasse l'algorithme de recherche aléatoire. Il est à remarquer que l'algorithme de planification uniforme dépasse l'algorithme UCT modifié pour $n \geq 21844$.

Le deuxième graphique 4.24 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 9$ et une profondeur $d \in [0, 5]$. Nous pouvons observer que les performances globales ont baissé hormis pour l'algorithme de recherche aléatoire dont les performances sont quasiment identiques. L'algorithme de planification optimiste dépasse l'algorithme UCT modifié pour $n \geq 7380$ alors que l'algorithme de planification uniforme n'arrive pas à obtenir des performances qui lui sont supérieures pour $n = 597870$.

Le troisième graphique 4.25 représente la moyenne de la somme des récompenses pour un facteur de branchement $K = 25$ et une profondeur $d \in [0, 3]\cdot$ Comme précédemment, les performances de l'algorithme de recherche aléatoire dominent celles des trois autres algorithmes, domination renforcée par un facteur de branchement élevé. L'algorithme de planification optimiste obtient des performances quasiment identiques à celles de l'algorithme UCT modifié, l'algorithme de planification uniforme étant légèrement en retrait.

Le problème du double cart-pole met un peu plus en exergue encore l'influence du facteur de branchement sur les performances des algorithmes de planification uniforme et optimiste et de l'algorithme UCT modifié ainsi que sur la relative indépendance au facteur de branchement de l'algorithme de recherche aléatoire.

Les résultats obtenus sont encourageant mais montrent une forte dépendance à la définition de la fonction récompense. En effet, si celle-ci est très directrice alors l'algorithme de planification optimiste obtient de très bon résultats expérimentaux. Cependant si la fonction récompense n'est pas directrice, l'inverse se produit. Ce comportement est cependant logique car si nous prenons comme exemple une fonction récompense qui ne donne la récompense maximum 1 que lorsqu'un état final a été atteint et partout ailleurs la récompense minimum 0, l'algorithme de planification optimiste n'aura d'autre choix dans ce cas précis que de construire un arbre des possibilités uniforme. La différence de comportement avec l'algorithme UCT modifié vient du fait que celui-ci n'utilise pas une vrai borne supérieure sur la somme des récompenses mais une heuristique. Ainsi, dans des cas problématiques comme celui du mountain car, l'algorithme UCT modifié obtient de meilleures performances que l'algorithme de planification optimiste.

Nous avons remarqué dans la section précédente que, malgré le fait que la borne supérieure sur le regret de l'algorithme de planification optimiste n'est pas pire que celle de l'algorithme de planification uniforme, les performances de l'algorithme de planification optimiste n'étaient pas toujours aussi bonnes que celles de l'algorithme de planification uniforme. Cependant au travers des

expérimentations effectuées, les performances de l'algorithme de planification optimiste furent toujours au moins aussi bonnes que celles de l'algorithme de planification uniforme. Ceci est dû en partie au fait que le choix arbitraire d'une action est identique dans l'implémentation pour les deux algorithmes de planification.

Cette différence de performances entre l'algorithme de planification optimiste et l'algorithme de planification uniforme peut être aussi dû au facteur de branchement κ de l'arbre $\mathcal{T}_\infty$. En effet, celui-ci est probablement inférieur au facteur de branchement K dans de nombreuses instances des problèmes rencontrés au cours nos expérimentations favorisant alors l'algorithme de planification optimiste à avoir de meilleures performances que l'algorithme de planification uniforme. Cependant il est à remarquer que l'algorithme de planification optimiste — tout comme l'algorithme de planification uniforme et l'algorithme UCT modifié — est influencé dans ses performances par le facteur de branchement K ce qui n'est pas surprenant au vu de la borne supérieure sur le regret des deux algorithmes de planification.

Une caractéristique intéressante de l'algorithme de planification optimiste — valable aussi pour l'uniforme — est la possibilité de réutilisation dans le pas de temps suivant du système à contrôler d'un sous-ensemble de l'arbre des possibilités étendu. En effet, si l'état successeur du système à contrôler obtenu par l'application de l'action retournée par l'algorithme de planification optimiste est identique à celui présent dans l'arbre, alors le sous-arbre issu de cette action et ayant cet état successeur comme racine peut être réutilisé pour faciliter les calculs.

Ceci est possible car la borne b_i est définie récursivement sur les nœuds enfants $\mathcal{C}(i)$ du nœud i et que le chemin pour atteindre le nœud i depuis la racine est unique et donc commun à tous les nœud du sous-arbre issu du nœud i. Ainsi, si l'état associé à la racine d'un sous-arbre devient l'état courant du système à contrôler, l'arbre étendu l'ayant pour racine sera identique au précédent sous-arbre pour la même quantité de ressources computationnelles. Il est alors avantageux de garder le sous-arbre et de remettre à jour la borne associée à chacun de ses nœuds.

Plus formellement, soit $k \in \{1, \ldots, K\}$, l'action retournée par l'algorithme de planification optimiste au pas de temps t du système à contrôler. Si le nouvel état du système à contrôler au pas de temps $t+1$ est identique à l'état associé au nœud k de l'arbre étendu $\mathcal{T}_t$, alors l'arbre étendu $\mathcal{T}_{t+1}$ peut être initialisé par le sous-arbre issu du nœud k avec les bornes remises à jour telles que $\forall i \in \mathcal{T}_{t+1}, b_i = \frac{b_i - r(x_t,k)}{\gamma}$ avec x_t l'état du système à contrôler au pas de temps t.

Cette approche est particulièrement avantageuse, par exemple, lorsqu'un appel au modèle génératif est très couteux et que les ressources computationnelles sont exprimées par une contrainte de temps. En effet, cela permet d'augmenter le nombre d'appels au modèle génératif effectués sur le même laps de temps. Cette idée de réutilisation des calculs précédents est aussi

présente dans le cadre de l'algorithme A* au travers de l'algorithme Lifelong Planning A* (LPA*) de [Koenig et al., 2004b] où les informations obtenues lors de planifications précédentes sont utilisées pour accélérer la convergence vers le chemin le plus cours dans les planifications suivantes.

Dans la même optique d'optimisation du nombre d'appels au modèle génératif sur un laps de temps donnée, la parallélisation de l'algorithme de planification optimiste est une approche à explorer. D'autant plus que, de nos jours, la multiplication des microprocesseurs multi-cœur incite à la parallélisation des algorithmes pour en tirer profit. Dans le cadre du jeu de go, l'algorithme UCT a fait l'objet de travaux visant à le paralléliser efficacement pour obtenir de meilleures performances de jeu.

Différentes méthodes ont été utilisées comme, par exemple, exécuter plusieurs instances de l'algorithme UCT et partager, au cours de l'exécution ou seulement lorsque les ressources computationnelles sont épuisées, des statistiques sur les différentes actions présentes à la racine — [Cazenave and Jouandeau, 2007], [Chaslot et al., 2008], [Gelly et al., 2008] — ou sur des sous-ensembles des nœuds de chaque arbre — [Bourki et al., 2010]. D'autres méthodes utilisent le même arbre mais contraigne son exploration soit par des *mutex* globaux ou locaux soit par une perte virtuelle — virtual loss — influençant l'exploration et ne concentrant ainsi pas l'exploration dans les mêmes parties de l'arbre — [Chaslot et al., 2008]. [Yoshizoe et al., 2011] propose de distribuer l'arbre et d'utiliser un algorithme de répartition de la charge de travail nommé Transposition Driven Scheduling (TDS) pour parcourir et mettre à jour l'arbre. Enfin [Cazenave and Jouandeau, 2007] et [Chaslot et al., 2008] se proposent de paralléliser l'évaluation d'une feuille et d'augmenter ainsi la qualité de son estimation.

Ses méthodes reposent principalement sur l'aspect stochastique des dynamiques du système à contrôler permettant ainsi de contourner l'aspect séquentiel de l'algorithme UCT. Dans notre cas, les dynamiques des systèmes à contrôler que nous envisageons sont déterministes, limitant ainsi la transposition de ces diverses méthodes. Cependant, nous pouvons nous en inspirer.

Une première méthode serait de lancer n instances de l'algorithme de planification optimiste avec, comme états initiaux, les états associés aux n feuilles à étendre d'un arbre étendu par l'algorithme de planification uniforme après $(n-1)/(K-1)$ nœuds étendus — il faut étendre $(n-1)/(K-1)$ nœuds pour obtenir n feuilles à étendre. La somme des récompenses dépréciées maximum est retournée par les n instances de l'algorithme de planification optimiste et est utilisée pour remettre à jour la somme des récompenses dépréciées de la racine de l'arbre étendu par l'algorithme de planification uniforme jusqu'à chacune des feuilles. L'action associée à la somme des récompenses dépréciées maximum est retournée.

Une seconde méthode serait d'effectuer à l'avance et en parallèle les appels au modèle génératif sur les états associés aux feuilles de l'arbre étendu

par l'algorithme de planification optimiste. En effet, chacune des feuilles possédant une borne, elles peuvent être ordonnées en une liste décroissante. Lorsqu'une feuille est sélectionnée pour être étendu, les états successeurs et les récompenses issus d'appels au modèle génératif effectués par anticipation sont utilisés. Il est à noter qu'il peut être aussi intéressant de paralléliser les K appels au modèle génératif issus d'un même état. Contrairement à la première méthode, celle-ci ne change pas la répartition de l'effort computationnel mais requière significativement plus de communication entre les différents processus — transmission des états et des résultats.

Dans le chapitre suivant, nous allons voir le cas où l'espace d'action n'est plus discret mais continu. Même si une solution trivial serait de discrétiser l'espace d'action continu pour obtenir un espace d'action discret, les expérimentations que nous venons de mener montrent que cette solution n'est pas viable, un nombre élevé d'actions dégradant sensiblement les performances des différents algorithmes.

Chapitre 5

Planification Optimiste avec espace d'action continu

Nous avons présenté dans le chapitre précédent l'algorithme de planification optimiste dans le cas où l'espace d'action est discret. Dans ce chapitre, nous présentons deux algorithmes de planification avec espace d'action continu. L'un a une approche inspirée de l'optimisation lipschitzienne en plus d'un comportement optimiste. L'autre propose une approche intuitive et plus simple à mettre en œuvre.

Le principal défi présenté par un espace d'action continu est lié à sa discrétisation. Une solution simple serait de discrétiser l'espace uniformément sans tenir compte des récompenses potentielles et de construire ainsi un arbre des possibilités uniforme. Cependant si les fonctions valeur et récompense possèdent une régularité, il serait alors opportun de l'exploiter et d'adapter la discrétisation en conséquence et ce pour chaque état rencontré au cours de la construction de l'arbre des possibilités.

Un problème vient s'ajouter à la discrétisation et est lié au dilemme exploration-exploitation. Lors de la création de l'arbre des possibilités, nous avons à choisir à chaque itération entre élargir l'arbre — discrétiser l'espace d'action plus finement — ou prolonger une séquence d'actions. Le premier choix correspond à l'exploration de l'arbre des possibilités car il ouvre accès à de nouvelles séquences d'actions autres que celles déjà explorées. Le deuxième choix correspond à l'exploitation car il affine l'estimation de la qualité de la première action prise dans la séquence prolongée. Il convient donc de concilier ces deux aspects en fonction du problème et des hypothèses considérées.

Ce problème peut aussi être vu comme la recherche dans l'espace des séquences infinies d'actions de la séquence maximisant la somme des récompenses dépréciées. Si l'espace d'action est continu et borné alors l'espace des séquences infinies d'actions peut être représenté comme un hypercube de dimension infinie dont chaque dimension est associée à une action dans une séquence. Nous nous intéressons donc à discrétiser cet espace de dimension

infinie et ce pour trouver le point maximisant la somme des récompenses dépréciées. Il est à noter cependant que seule la coordonnée dans la première dimension de ce point nous intéresse car seule la première action de la séquence d'actions maximisant la somme des récompenses dépréciées nous intéresse.

La recherche d'une politique optimale dans le cas d'espace d'action continu est très présente dans la littérature. Nous pouvons citer en premier l'algorithme HOOT de [Mansley et al., 2011] présenté dans le chapitre 3 à la page 39 adapté pour la planification comme nous l'avons défini précédemment. Cependant la plus grande partie des approches envisagées dans la littérature utilise des approximations pour apprendre une politique optimale — [Baird and Klopf, 1993], [Gaskett et al., 1999], [Millán et al., 2002], [Sallans et al., 2004], [Lazaric et al., 2007]. Une autre approche envisagée par [Pazis and Lagoudakis, 2009b] et [Pazis and Lagoudakis, 2009a] consiste à augmenter ou diminuer l'action précédente en fonction d'une politique binaire, l'espace d'état étant augmenté pour contenir la dite action précédente. La modification de l'action précédente peut se faire de deux manières différentes ; soit une modification par pas de temps et par rapport à la politique binaire avec une variation de l'amplitude de la modification en fonction de la nature des modifications précédentes ; soit une itération des modifications en fonction d'une politique binaire jusqu'à atteindre une certaine finesse. L'idée de modifier l'action précédente par incrément positif ou négatif est intéressante car elle permet d'obtenir des actions discrètes et pourrait être adapté à l'algorithme de planification optimiste.

5.1 Planification Lipschitzienne

Dans cette section, nous présentons un algorithme de planification avec espace d'action continu s'inspirant de l'approche présente dans le domaine de l'optimisation lipschitzienne en ce sens que nous nous servons d'hypothèses lipschitziennes pour guider la discrétisation de l'espace des séquences infinies d'actions. Ainsi nous définissons une borne sur une subdivision de l'espace des séquences infinies d'actions et choisissons de discrétiser plus finement la subdivision possédant la borne maximum.

Cette approche est à mettre en lien avec la planification optimiste présentée dans le chapitre précédent. En effet, celle-ci peut être vu comme l'exploration d'un espace de séquences infinies d'actions dont la discrétisation est fixée par le facteur de branchement. Dans le cas présent, la discrétisation n'est pas fixée et sera effectuée de manière adaptative.

Ainsi nous présenterons les hypothèses sur les problèmes envisagés depuis lesquelles sont dérivées une borne sur un ensemble de sous-espaces représentant une séquence d'actions. De cette borne, possédant par ailleurs un aspect optimiste, découle l'algorithme de planification lipschitzienne.

5.1.1 Description

Nous allons dans un premier temps décrire les notations et formaliser le problème. Puis nous définirons une borne sur un sous-ensemble de sous-espaces de l'espace des séquences infinies d'actions A^∞. Enfin nous présenterons l'algorithme de planification lipschitzienne.

Notations et formalisation du problème

Comme précédemment nous souhaitons contrôler un système déterministe mais dont les actions sont définies sur un espace continu et non discret. Nous supposons avoir accès à un modèle génératif $f : X \times A \to X$ avec X un espace d'état quelconque et A un espace d'action continu tel que $x_{t+1} = f(x_t, a_t)$, x_{t+1} étant l'état successeur lorsque l'action a_t est effectuée dans l'état x_t. Nous supposons aussi avoir accès à la fonction récompense $r : X \times A \to [0, 1]$.

Nous définissons $\mathbf{a} \in A^n$ comme une séquence de n actions et pour tout $0 \leq t \leq n-1 \leq +\infty$ et $\mathbf{a} \in A^n$, $r_t(\mathbf{a}) = r(x_t, a_t)$ la récompense obtenue à l'instant t le long de la trajectoire commençant en l'état x_0 et appliquant la séquence d'actions $\mathbf{a}$. Nous définissons A^∞ comme l'espace des séquences infinies d'actions appartenant à A.

Notre objectif est de trouver une action $a \in A$ ou une séquence d'actions $\mathbf{a} \in A^n$ aussi proche que possible de l'action optimale $a^* \in A$ ou de la séquence optimale $\mathbf{a}^* \in A^\infty$. Cette action a ou séquence d'actions $\mathbf{a}$ est retournée par un algorithme de sélection d'action ayant accès au modèle génératif mais étant contraint par des ressources computationnelles limitées et inconnues. Dans le reste du chapitre, nous ne parlerons plus que d'une action a retournée par l'algorithme de sélection d'action, le cas d'une séquence d'actions $\mathbf{a}$ étant une extension directe.

Pour ce faire, l'algorithme de planification lipschitzienne explore A^∞ en le discrétisant en sous-espaces de dimension grandissante et retourne l'action menant à la somme des récompenses dépréciées la plus élevée rencontrée pendant ce processus lorsque les ressources computationnelles sont épuisées.

Il est à noter que A^∞ peut être aussi vu comme l'espace des politiques étant donné un état. En effet, dans le cas déterministe, une politique est une séquence infinie d'actions définie par l'état initial. Ainsi, étant donné un état initial, A^∞ représente l'ensemble des politiques applicables depuis cet état.

Pour tout état initial $x_0 \in X$, nous définissons la fonction valeur

$$V^*(x_0) \stackrel{\text{def}}{=} \max_{\mathbf{a}=(a_0,a_1,\dots)} \sum_{t \geq 0} \gamma^t r(x_t, a_t).$$

Nous réécrivons la fonction valeur V comme fonction d'une séquence finie de n actions $\mathbf{a}$ telles que $V^*(x_0) = \max_{\mathbf{a} \in A^n} V(\mathbf{a})$. Ainsi pour tout $\mathbf{a} \in A^n$,

nous définissons $V(\mathbf{a})$ tel que :

$$V(\mathbf{a}) = \sum_{t=0}^{n-1} \gamma^t r_t(\mathbf{a}) + \max_{\mathbf{a}'=(a_n,a_{n+1},\dots)\in A^\infty} \sum_{t\geq n} \gamma^t r_t(\mathbf{a}+\mathbf{a}') \qquad (5.1)$$

avec $(\mathbf{a}+\mathbf{a}') \in A^{p+q}$ la concaténation de $\mathbf{a} \in A^p$ et $\mathbf{a} \in A^q$ et $\gamma \in [0,1[$.

Notre algorithme a pour objectif de retourner une action $a \in A$ proche de l'optimale maximisant ainsi $V(a)$. Cet maximisation représente une tâche non triviale de part la présence de l'opérateur max. En effet l'espace d'action A étant continu, l'explorer requière d'utiliser une méthode de discrétisation. Une approche simple serait de discrétiser uniformément l'espace d'action. Ces actions discrétisées seraient ensuite utilisées pour construire un arbre des possibilités uniforme et ce jusqu'à l'épuisement des ressources computationnelles.

Le principal inconvénient de cette méthode est que, puisque la discrétisation est fixée à l'avance, elle ne s'adapte pas à la répartition des récompenses sur l'espace des séquences infinies d'actions A^∞. En effet, si la répartition était connue à l'avance, nous aimerions discrétiser plus finement l'espace d'action là où les récompenses sont les plus élevées tout en prenant en compte le facteur de dépréciation et ainsi privilégier la discrétisation des espaces d'action proches de l'état initial x_0 en place et lieu de ceux plus profonds dans les séquences d'actions.

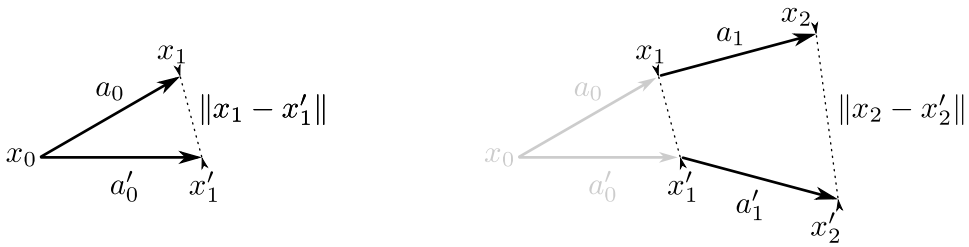
L'algorithme que nous proposons discrétise l'espace A^∞ en séquences d'actions de longueur grandissante, chacune des actions étant associée avec un sous-espace de A l'incluant et chacune des séquences étant un sous-ensemble formé de sous-espaces. Le processus de discrétisation est dirigé par une borne supérieure sur chaque sous-ensemble de sous-espaces. Cette borne supérieure représente la somme des récompenses dépréciées maximale atteignable au sein de ce sous-ensemble de sous-espaces. Lorsque les ressources computationnelles sont épuisées, notre algorithme retourne l'action a qui est la première action de la séquence d'actions possédant la somme des récompenses dépréciées la plus élevée parmi les séquences explorées.

Pour définir une borne supérieure sur un sous-ensemble de sous-espaces nous supposons l'hypothèse suivante à propos des récompenses :

Hypothèse 1. *Pour toutes séquences infinies d'actions $\boldsymbol{a}$ et $\boldsymbol{a}' \in A^\infty$, nous supposons que pour tout $t \geq 0$, $|r_t(\boldsymbol{a}) - r_t(\boldsymbol{a}')| \leq \sum_{i=0}^{t} L^{t-i+1}\|a_i - a'_i\|$, pour une constante $L \geq 0$.*

Cette hypothèse est vérifiée lorsque la fonction transition et la fonction récompense sont lipschitziennes, c'est-à-dire qu'il existe $L_{f_x}, L_{f_a}, L_{r_x}$ et $L_{r_a} \in \mathbb{R}^+$ tels que

$$\begin{aligned} \|f(x_t,a_t) - f(x'_t,a'_t)\| &\leq L_{f_x}\|x_t - x'_t\| + L_{f_a}\|a_t - a'_t\|, \\ |r_t(\mathbf{a}) - r_t(\mathbf{a}')| &\leq L_{r_x}\|x_t - x'_t\| + L_{r_a}\|a_t - a'_t\| \end{aligned}$$



(a) Première action. (b) Deuxième action.

FIGURE 5.1 – Évaluation de la distance entre deux états issus de deux séquences d'actions.

et lorsque $L = \max(L_{f_x}, L_{f_a}, L_{r_x}, L_{r_a}$.

L'intuition de l'hypothèse 1 est décrite par la figure 5.1. À partir de l'état x_0 et en lui appliquant l'action a_0 et l'action a'_0, nous obtenons deux nouveaux états x_1 et x'_1 — voir la figure 5.1a — lesquels étant donné l'hypothèse lipschitzienne sur la fonction transition vérifient

$$\|x_1 - x'_1\| \leq L_{f_x}\|x_0 - x_0\| + L_{f_a}\|a_0 - a'_0\| = L_{f_a}\|a_0 - a'_0\|.$$

De la même façon, à partir des deux états x_1 et x'_1, nous appliquons respectivement les actions a_1 et a'_1 et obtenons deux nouveaux états x_2 et x'_2 — voir la figure 5.1b — lesquels vérifient

$$\|x_2 - x'_2\| \leq L_{f_x}\|x_1 - x'_1\| + L_{f_a}\|a_1 - a'_1\|.$$

Grâce à la première égalité, nous pouvons écrire

$$\|x_2 - x'_2\| \leq L^2\|a_0 - a'_0\| + L\|a_1 - a'_1\|$$

pour $L = \max(L_{f_x}, L_{f_a})$.

En itérant ce procédé, nous pouvons déduire que pour $t > 0$

$$\|x_t - x'_t\| \leq \sum_{i=0}^{t-1} L^{i+1}\|a_{t-1-i} - a'_{t-1-i}\|$$

pour $L = \max(L_{f_x}, L_{f_a})$.

Nous avons aussi énoncé comme condition que la fonction récompense est lipschitzienne ainsi pour tout $t > 0$

$$\begin{array}{rcl} |r_t(\mathbf{a}) - r_t(\mathbf{a}')| & \leq & L_{r_x}(\sum_{i=0}^{t-1} L^{i+1}\|a_{t-1-i} - a'_{t-1-i}\|) + L_{r_a}\|a_t - a'_t\| \\ & \leq & \sum_{i=0}^{t} L^{t-s+1}\|a_i - a'_i\| \end{array}$$

pour $L = \max(L_{f_x}, L_{f_a}, L_{r_x}, L_{r_a})$.

Définition 1. Nous définissons la semimétrique l sur l'espace A^∞ telle que

$$l(\mathbf{a}, \mathbf{a}') = \sum_{t \geq 0} \gamma^t \min(1, \sum_{i=0}^{t} L^{t-i+1} \|a_i - a'_i\|$$

avec $\mathbf{a}, \mathbf{a}' \in A^\infty$, une constante $L \geq 0$ et $\gamma \in [0, 1[$.

Soit $T(\mathbf{a}, \mathbf{a}')$ le plus grand entier $t \geq 0$ tel que $\sum_{i=0}^{t} L^{t-i+1} \|a_i - a'_i\| \leq 1$ avec $\mathbf{a}, \mathbf{a}' \in A^\infty$.

Nous pouvons réécrire la semimétrique l telle que

$$l(\mathbf{a}, \mathbf{a}') = \sum_{t=0}^{T(\mathbf{a}, \mathbf{a}')} \gamma^t \sum_{i=0}^{t} L^{t-i+1} \|a_i - a'_i\| + \frac{\gamma^{T(\mathbf{a}, \mathbf{a}')+1}}{1-\gamma} \tag{5.2}$$

avec $\mathbf{a}, \mathbf{a}' \in A^\infty$, une constante $L \geq 0$ et $\gamma \in [0, 1[$.

Proposition 2. *À partir de l'équation (5.1) et de l'hypothèse 1, nous avons la propriété que pour tout* $\boldsymbol{a}$ *et* $\mathbf{a}' \in A^\infty$

$$|V(\boldsymbol{a}) - V(\boldsymbol{a}')| \leq l(\boldsymbol{a}, \boldsymbol{a}') \tag{5.3}$$

Borne sur un sous-ensemble de sous-espaces de A^∞

Nous allons maintenant à partir de la semimétrique l définir une borne supérieure sur un sous-ensemble de sous-espaces de A^∞. Nous définissons le partitionnement de A^∞ — voir la figure 5.2 — en considérant n sous-ensembles $\mathcal{A}_i \subset A^\infty$ où chaque sous-ensemble $\mathcal{A}_i$ est constitué par une séquence finie de sous-espaces $(b_0^i, \ldots, b_{n_i}^i)$ où $b_t^i \subset A$ telle que :

$$\mathcal{A}_i = \left\{ \mathbf{a} \in A^\infty | \forall t \in \{0, \ldots, n_i\}, a_t^i \in b_t^i \right\}$$

est l'ensemble des séquences infinies d'actions $\mathbf{a}^i$ appartenant à $\mathcal{A}_i$.

Sur la figure 5.2, l'espace A^∞ — représentée ici comme un hypercube — est partitionné selon la première action — celle-ci étant la première dimension de A^∞ — en deux sous-ensembles $\mathcal{A}_0$ et $\mathcal{A}_1$ contenant respectivement le sous-espace b_0^0 et le sous-espace b_0^1. L'espace A^∞ est ensuite partitionné selon la deuxième action au sein du sous-ensemble A_0. Un nouveau sous-ensemble $\mathcal{A}_2$ est créé et contient deux sous-espaces b_0^2 et b_1^2. De plus, le sous-espace b_1^0 est ajouté au sous-ensemble $\mathcal{A}_0$.

Comme nous l'avons vu précédemment, la semimétrique l est une borne supérieure sur la différence entre la valeur d'une séquence d'actions $\mathbf{a}$ et d'une séquence d'actions $\mathbf{a}'$ — autrement dit, c'est une borne supérieure sur la différence entres les sommes de récompenses dépréciées obtenues en suivant les deux séquences d'actions depuis l'état initial. Si l'on fixe une séquence d'actions ${\mathbf{a}_c}^i$ pour un sous-ensemble $\mathcal{A}_i$, grâce à l'équation (5.3), nous avons

$$\max_{\mathbf{a} \in \mathcal{A}_i} V(\mathbf{a}) \leq V({\mathbf{a}_c}^i) + \max_{\mathbf{a} \in \mathcal{A}_i} l({\mathbf{a}_c}^i, \mathbf{a}) \tag{5.4}$$

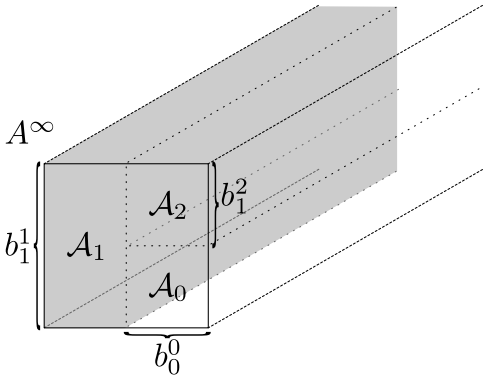


FIGURE 5.2 – Un exemple de partitionnement de A^∞.

D'après la définition de la semimétrique l, $\max_{\mathbf{a}\in\mathcal{A}_i} l({\mathbf{a}_c}^i, \mathbf{a}) = l({\mathbf{a}_c}^i, \mathbf{a}')$ avec $\mathbf{a}' = \arg\max_{\mathbf{a}\in\mathcal{A}_i} \sum_{t\geq 0} \|a_{ct}^i - a_t^i\|$.

À partir de l'équation (5.2) et de l'équation (5.4), nous pouvons écrire que :

$$\begin{array}{rcl} \max_{\mathbf{a}\in\mathcal{A}_i} V(\mathbf{a}) & \leq & \sum_{t=0}^{n_i} \gamma^t r_t({\mathbf{a}_c}^i) + \frac{\gamma^{n_i+1}}{1-\gamma} + \max_{\mathbf{a}\in\mathcal{A}_i} l({\mathbf{a}_c}^i, \mathbf{a}) \\ & \leq & \sum_{t=0}^{n_i} \gamma^t r_t({\mathbf{a}_c}^i) + \frac{\gamma^{n_i+1}}{1-\gamma} \\ & & + \max_{\mathbf{a}\in\mathcal{A}_i} \sum_{t=0}^{T({\mathbf{a}_c}^i,\mathbf{a})} \gamma^t \sum_{j=0}^{t} L^{t-j+1} \|a_{cj}^i - a_j\| \\ & & + \frac{\gamma^{T({\mathbf{a}_c}^i,\mathbf{a})+1}}{1-\gamma}. \end{array}$$

Nous définissons δ_t^i la taille d'un sous-espace b_t^i du sous-ensemble $\mathcal{A}_i$ telle que

$$\delta_t^i = \begin{cases} \max_{a\in b_t^i} \|a_{ct}^i - a\| & \text{pour } t \in \{0, \ldots, n_i\}, \\ +\infty & \text{pour } t > n_i. \end{cases}$$

Nous définissons aussi T_i comme étant le plus grand entier $t \leq n_i$ tel que $r_t({\mathbf{a}_c}^i) + \sum_{j=0}^{t} L^{t-j+1}\delta_j^i \leq 1$.

À partir des définitions précédentes, nous pouvons définir une borne supérieure B_i sur un sous-ensemble $\mathcal{A}_i$ telle que

$$B_i = \underbrace{\sum_{t=0}^{T_i} \gamma^t r_t\left({\mathbf{a}_c}^i\right)}_{1} + \underbrace{\overbrace{\sum_{t=0}^{T_i} \sum_{j=0}^{t} L^{t-j+1}\delta_j^i}^{3} + \overbrace{\frac{\gamma^{T_i+1}}{1-\gamma}}^{4}}_{2}.$$

La partie 1 de la borne B_i est la somme des récompenses dépréciées obtenues au sein du sous-ensemble $\mathcal{A}_i$ en suivant la séquence d'actions $\mathbf{a}_c^i$ sur n_i+1 pas de temps. Elle contribue à évaluer la qualité réelle du sous-ensemble $\mathcal{A}_i$. La partie 2 de la borne évalue la qualité potentielle du sous-ensemble $\mathcal{A}_i$ et ce de manière optimiste. Elle représente la différence maximum entre la

somme des récompenses dépréciées de la partie 1 et tout autre séquence d'actions appartenant au sous-ensemble $\mathcal{A}_i$. La partie 3 est la différence maximum obtenue en déviant de la séquence d'action ${\mathbf{a}_c}^i$ au sein du sous-ensemble $\mathcal{A}_i$. Ainsi, si la taille des sous-espaces b_t^i est élevée ou si la constante de Lipschitz L est grande, dévier de la séquence d'actions ${\mathbf{a}_c}^i$ permettra d'obtenir potentiellement une somme des récompenses dépréciées plus élevée que celle obtenue par la partie 1. Cette partie favorise la recherche en largeur et donc la discrétisation des sous-espaces existants. La partie 4 est la somme maximum des récompenses dépréciées atteignable dans les dimensions encore non-explorées de A^∞. Cette partie favorise la recherche en profondeur et donc l'ajout de nouveaux sous-espaces au sous-ensemble $\mathcal{A}_i$.

D'un point de vue plus général, nous avons défini une semimétrique sur l'espace A^∞ des séquence infinies d'actions nous donnant une indication de la distance entre deux séquences infinies d'actions. Cette distance est la différence maximum entre les deux sommes de récompenses dépréciées. Cette semimétrique l peut être utilisée dans tout algorithme d'optimisation optimiste en vu de diriger le partitionnement de l'espace A^∞.

Algorithme

Nous avons présenté la borne dans un cadre générique où aucune hypothèse n'a été formulée sur la forme des sous-espaces ou sur la façon dont ils sont coupés. Nous présentons maintenant l'algorithme de planification lipschitzienne utilisant la borne B_i ainsi que la stratégie de discrétisation dans le cas où l'espace d'action continu A est représenté par un hypercube.

Nous supposerons que l'espace d'action continu A est mis à l'échelle pour prendre la forme d'un hypercube dont chaque côté est de largeur 1. Chaque action ${a_c}_t^i$ dans chaque séquence d'actions ${\mathbf{a}_c}^i$ est le centre du sous-espace associé b_t^i. Ainsi pour garder l'action associée ${a_c}_t^i$ au centre du sous-espace b_t^i — et de fait ne pas gaspiller les ressources computationnelles consommées par l'utilisation du modèle génératif — lorsqu'un sous-espace b_t^i est discrétisé, il est coupé en trois. Dans le cas où le nombre de dimensions de l'espace d'action continu A est strictement supérieur à 1, lorsqu'un sous-espace b_t^i est discrétisé, son plus grand côté est trisecté.

L'idée générale de l'algorithme de planification Lipschitzienne est de sélectionner à chaque itération le sous-ensemble $\mathcal{A}_i$ possédant la borne B_i la plus élevée. Une fois le sous-ensemble sélectionné, la discrétisation qui contribue à la diminution la plus forte la borne est appliquée. Ainsi, si un sous-espace b_t^i est discrétisé, sa taille δ_t^i diminue, la partie 3 de la borne diminue en fonction de la constante de Lipschitz L et de la distance du sous-espace à l'état initial et contribue ainsi à l'exploration en largeur de l'arbre des possibilités. À l'inverse, si un sous-espace est ajouté au sous-ensemble, la partie 4 de la borne diminue en fonction du facteur de dépréciation γ et du nombre de sous-espaces appartenant au sous-ensemble et participe à

l'exploration en profondeur de l'arbre des possibilités. Ainsi en fonction de la constante de Lipschitz L et du facteur de dépréciation γ, l'exploration en largeur ou en profondeur sera plus ou moins accentuée.

Algorithme 5.1 Planification Lipschitzienne

Initialiser x_0 à l'état courant du système
Initialiser $\mathcal{A}_0$
Initialiser B_0
while Une condition d'arrêt n'est pas vraie **do**
  Sélectionner le sous-ensemble $\mathcal{A}_{i_{\max}}$ qui maximise $B_{i_{\max}}$ pour $i_{\max} \geq 0$
  Calculer le plus grand entier $T \leq n_{i_{\max}}$ tel que $r_T({\mathbf{a}_c}^{i_{\max}}) + \sum_{j=0}^{T} L^{T-j+1}\delta_j^{i_{\max}} \leq 1$
  Sélectionner le sous-espace $b_{t_{\min}}^{i_{\max}} \in \mathcal{A}_{i_{\max}}$ qui minimise $B_{i_{\max}}$ avec $t_{\min} \in \{0, \ldots, T\}$
  if T est égale à $n_{i_{\max}}$ **then**
    Sélectionner entre discrétiser $b_{t_{\min}}^{i_{\max}}$ ou ajouter un nouveau sous-espace $b_{n_{i_{\max}}+1}^{i_{\max}}$ selon leur borne $B_{i_{\max}}$ respective résultante
  end if
  if La discrétisation est d'ajouter un nouveau sous-espace à $\mathcal{A}_{i_{\max}}$ **then**
    Ajouter le sous-espace $b_{n_{i_{\max}}+1}^{i_{\max}}$ au sous-ensemble $\mathcal{A}_{i_{\max}}$
    Étendre la séquence d'actions ${\mathbf{a}_c}^{i_{\max}}$ avec une action supplémentaire
    Utiliser le modèle génératif pour calculer $r_{n_{i_{\max}}+1}({\mathbf{a}_c}^{i_{\max}})$
    Calculer le nouvelle borne $B_{i_{\max}}$
    Incrémenter $n_{i_{\max}}$ de un
  else
    Trisecter $b_{t_{\min}}^{i_{\max}}$
    Créer les nouveaux sous-ensembles
    Utiliser le modèle génératif pour calculer les récompenses
    Calculer les nouvelles bornes
  end if
end while
return L'action a_{c0}^{i} telle que $\max_{i \geq 0} \sum_{t=0}^{n_i} \gamma^t r_t({\mathbf{a}_c}^i)$.

L'algorithme de planification lipschitzienne est élaboré en utilisant la borne B_i et le partitionnement de A^∞ définis précédemment — voir l'algorithme 5.1. À chaque itération et avec $i_{\max} \geq 0$, le sous-ensemble $\mathcal{A}_{i_{\max}}$ avec la plus grande borne supérieure $B_{i_{\max}}$ est sélectionné pour être discrétisé car étant le plus prometteur. $T_{i_{\max}}$ est alors calculé et utilisé pour limiter la discrétisation aux sous-espaces $b_t^{i_{\max}}$ avec $t \in \{0, \ldots, T_{i_{\max}}\}$. En effet, avant de discrétiser les sous-espaces les plus profonds, il nous faut discrétiser les plus proches de l'état initial x_0 ceux-ci ayant plus de poids sur la qualité de l'action finale retournée. En contre partie, cela fait converger $T_{i_{\max}}$ vers $n_{i_{\max}}$ pour permettre aux sous-espaces plus profonds d'être discrétiser pen-

dant les itérations suivantes mettant ainsi en place un procédé proche de l'*Iterative-Deepening*.

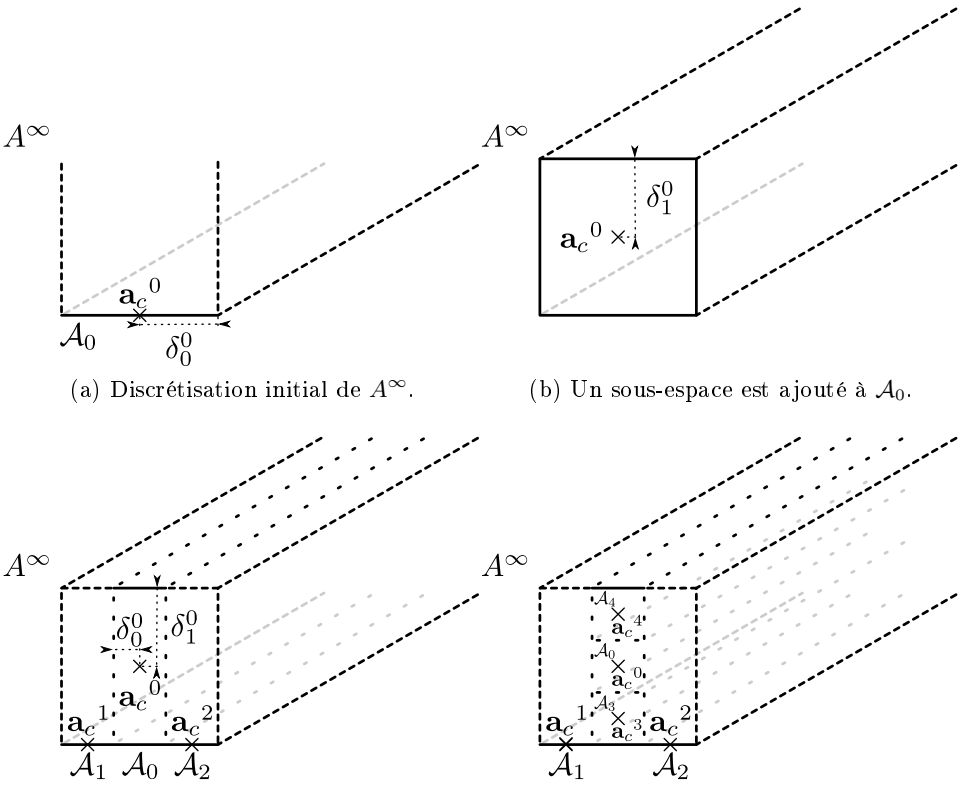
Pour chaque sous-espace $b_t^{i_{\max}}$ avec $t \in \{0, \ldots, T_{i_{\max}}\}$, la borne $B_{i_{\max}}$ est calculée dans le cas où le sous-espace $b_t^{i_{\max}}$ est choisi pour être discrétisé. Le sous-espace $b_{t_{\min}}^{i_{\max}}$ avec la plus petite borne $B_{i_{\max}}$ résultante est alors sélectionné avec $t_{\min} \in \{0, \ldots, T_{i_{\max}}\}$. Si $T_{i_{\max}}$ est égal à $n_{i_{\max}}$ alors le cas où un nouveau sous-espace est ajouté au sous-ensemble $\mathcal{A}_{i_{\max}}$ est pris en compte. La borne $B_{i_{\max}}$ est alors calculée avec le sous-espace $b_{n_{i_{\max}}+1}^{i_{\max}}$ ajouté au sous-ensemble — et en supposant que la récompense obtenue par l'application de l'action $a_{c_{n_{i_{\max}}+1}}^{i_{\max}}$ est 0 — et est comparée avec la borne calculée lorsque $b_{t_{\min}}^{i_{\max}}$ est trisecté. L'opération obtenant la plus petit borne est effectuée. Ajouter un nouveau sous-espace au sous-ensemble $\mathcal{A}_{i_{\max}}$ participe à son exploitation alors que discrétiser le sous-espace $b_{t_{\min}}^{i_{\max}}$ élargit l'exploration autour de la séquence d'actions $\mathbf{a}_c{}^{i_{\max}}$ en créant de nouvelles séquences d'actions.

Si un nouveau sous-espace $b_{n_{i_{\max}}+1}^{i_{\max}}$ est ajouté au sous-ensemble $\mathcal{A}_{i_{\max}}$, l'action $a_{c_{n_{i_{\max}}+1}}^{i_{\max}}$ est ajoutée à la séquence d'actions $\mathbf{a}_c{}^{i_{\max}}$ et le modèle génératif est utilisé pour calculer $r_{n_{i_{\max}}+1}(\mathbf{a}_c{}^{i_{\max}})$. $n_{i_{\max}}$ est incrémenté de 1 et la nouvelle borne $B_{i_{\max}}$ est calculée.

Sinon le sous-espace $b_{t_{\min}}^{i_{\max}}$ est trisecté selon son côté le plus grand et deux nouveaux sous-ensembles sont créés. Ces deux sous-ensembles sont constitués des sous-espaces $b_t^{i_{\max}}$ avec $t < t_{\min}$ et d'un des deux sous-espaces engendrés par la discrétisation de $b_{t_{\min}}^{i_{\max}}$. Les récompenses associées à ces deux nouveaux sous-espaces sont calculées par l'intermédiaire du modèle génératif et les bornes B sont calculées pour chacun des trois sous-ensembles. Il a noter que les futurs sous-espaces ajoutés à ces deux nouveaux sous-ensembles seront contraint par les sous-espaces $b_t^{i_{\max}}$ avec $t \in \{t_{\min}+1, \ldots, n_{i_{\max}}\}$ et ce pour éviter les chevauchements.

Lorsque les ressources computationnelles sont épuisées — ou toute autre condition atteinte — l'action $a_{c_0}^i$ avec $i \geq 0$ qui maximise $\sum_{t=0}^{n_i} \gamma^t r_t(\mathbf{a}_c{}^i)$ est retournée.

Un exemple de discrétisation itérative de A^∞ est présenté par la figure 5.3. La figure 5.3a montre le sous-ensemble initial $\mathcal{A}_0$ qui ne possède qu'un seul sous-espace b_0^0 avec l'action $a_{c_0}^0$ en son centre. Le sous-espace b_1^0 est ajouté au sous-ensemble $\mathcal{A}_0$ — voir la figure 5.3b — et l'action $a_{c_1}^0$ est ajouté à la séquence d'actions $\mathbf{a}_c{}^0$. Le sous-espace b_0^0 est trisecté en figure 5.3c et deux nouveaux sous-ensembles sont créés $\mathcal{A}_1$ et $\mathcal{A}_2$. Dans la figure 5.3d, le sous-ensemble $\mathcal{A}_0$ est sélectionné pour être discrétisé et le sous-espace b_1^0 est trisecté créant ainsi deux nouveaux sous-ensembles $\mathcal{A}_3$ et $\mathcal{A}_4$. Il est à noter que quand le sous-espace b_0^0 est trisecté, les séquences d'actions associées avec les sous-ensembles $\mathcal{A}_1$ et $\mathcal{A}_2$ sont limitées aux actions centrées dans les deux nouveaux sous-espaces — voir la figure 5.3c.



(a) Discrétisation initial de A^∞.

(b) Un sous-espace est ajouté à $\mathcal{A}_0$.

(c) Le sous-espace b_0^0 est trisecté.

(d) Le sous-espace b_1^0 est trisecté.

FIGURE 5.3 – Un exemple de discrétisation itérative de A^∞.

5.1.2 Expérimentations

Le problème du cart-pole

Le problème du cart-pole est identique à celui présenté dans le chapitre précédent — voir page 65 — à l'exception de l'espace d'action A qui est maintenant continu et défini par le domaine $[-10, 10]$.

L'expérimentation sur le problème du cart-pole consiste à calculer la moyenne sur 1000 états initiaux et pour différentes constantes de Lipschitz $L \in \{0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.75, 1\}$ et nombres $n \in \{100, 1000, 10000, 100000\}$ d'appels au modèle génératif de la somme des récompenses sur 100 pas de temps. Les états initiaux sont tirés uniformément sur le domaine $[-2, 2] \times [-5, 5] \times [1, 5.28] \times [-1, 1]$.

Nous pouvons observer sur le graphique 5.4 les résultats de la première expérimentation. L'axe horizontal de droite représente le nombre d'appels au modèle génératif sur une échelle logarithmique. L'axe horizontal de gauche représente les différentes constantes de Lipschitz. L'axe vertical représente la moyenne de la somme des récompenses. La courbe en rouge représente

l'algorithme de planification lipschitzienne et la courbe en verte représente l'algorithme de planification optimiste. L'algorithme de planification optimiste est expérimenté avec un facteur de branchement $K = 2$ et différents nombres $n \in \{100, 1000, 10000, 100000\}$ d'appels au modèle génératif. Il nous permet d'avoir un point de repère sur les performances obtenues.

Ainsi, nous voyons que les performances de l'algorithme de planification lipschitzienne sont fortement inférieures à celles obtenues par l'algorithme de planification optimiste. Nous pouvons remarquer que contrairement aux algorithmes précédents, les performances de l'algorithme de planification lipschitzienne ont tendance à baisser lorsque le nombre d'appels au modèle génératif augmente. Ce comportement est dû au fait que les constantes de Lipschitz utilisées ne sont pas des bornes supérieures pour nombre d'états initiaux. Ainsi, lorsque le nombre d'appels au modèle génératif augmente, une exploration plus profonde dû à une constante de Lipschitz trop faible peut prendre place biaisant le résultat. Ceci est à mettre en lien avec le problème nommé *look-ahead pathologies* décrit par [Péret and Garcia, 2004] et présenté dans le chapitre 3 à la page 28.

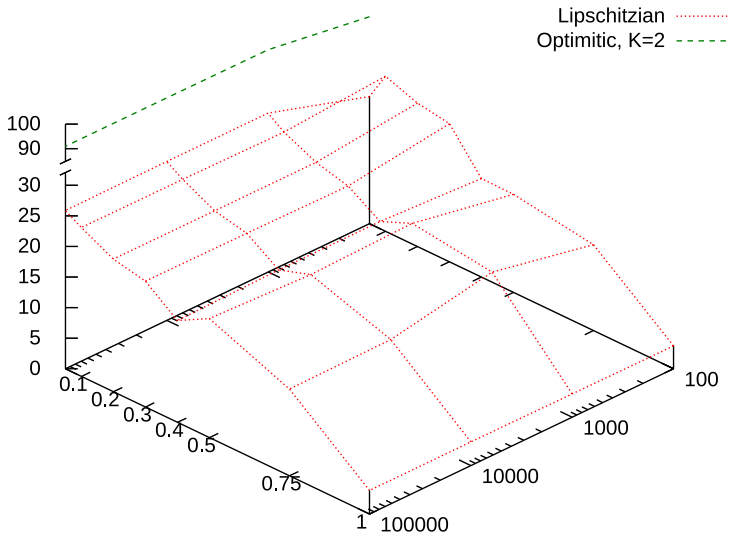


FIGURE 5.4 – Somme des récompenses pour le problème du cart-pole.

Le problème du cart-pole possède deux phases distinctes. La première consiste à augmenter la vitesse angulaire du mat par un effet de balancier. La deuxième phase est de maintenir en équilibre le mat. Intuitivement, nous pouvons déduire que la première phase possède une constante de Lipschitz faible alors que la deuxième phase possède une constante de Lipschitz plus

élevée.

Pour vérifier cette intuition, nous avons mené deux autre expérimentations. La première expérimentation consiste à tirer uniformément 1000 états initiaux sur le domaine $[-2, 2] \times [-0.5, 0.5] \times [2.64, 3.64] \times [-0.1, 0.1]$. Ces états initiaux sont proches de l'état de repos. Nous pouvons constater que contrairement à notre intuition première, les performances sont meilleures pour des constantes de Lipschitz situées entre 0.5 et 0.75 — voir le graphique 5.5. De plus, ceci nous indique que sous-estimer la constante de Lipschitz conduit à des performances réduites. Il est aussi à noter que les performances sont bien inférieures à celle de l'algorithme de planification optimiste. Nous pouvons en déduire que globalement, aucune constante de Lipschitz n'est à même d'obtenir une séquence d'actions conduisant à un mouvement de balancier par l'intermédiaire de l'algorithme de planification lipschitzienne.

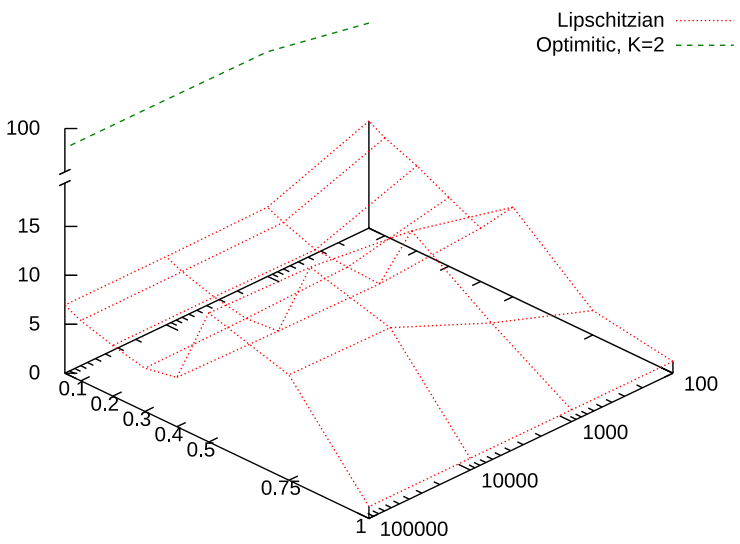


FIGURE 5.5 – Somme des récompenses pour le problème du cart-pole avec des états initiaux proche du repos.

La deuxième expérimentation consiste à tirer uniformément 1000 états initiaux sur le domaine $[-2, 2] \times [-0.5, 0.5] \times [0, 0.5] \cup [5.78, 6.26] \times [-0.1, 0.1]$. Ces états initiaux sont proches de l'équilibre. Nous pouvons constater sur le graphique 5.6 que les résultats sont globalement les mêmes pour chacun des constantes de Lipschitz utilisées à l'exception de $L = 1$. Ce comportement peut s'expliquer par le fait qu'étant proche de l'optimal, seules une discrétisation suffisante et une exploration en profondeur minimale sont requises pour garder la position d'équilibre. Cependant si la recherche se limite à la

discrétisation du premier sous-espace alors que le cart-pole se trouve en bord de piste, il est devient alors plus difficile d'éviter la sortie et d'obtenir des performances amoindries comme c'est le cas avec $L = 1$.

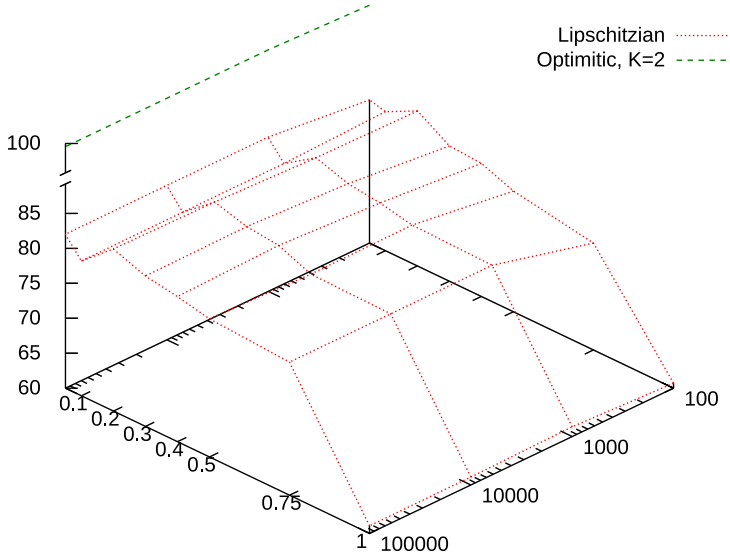


FIGURE 5.6 – Somme des récompenses pour le problème du cart-pole avec des états initiaux proche de l'équilibre.

La première phase du problème du cart-pole requière un contrôle du type *bang-bang* impliquant l'utilisation des deux points extrêmes de l'espace des actions. Pour qu'un algorithme de planification perçoive l'intérêt du mouvement de balancier, une profondeur minimum nécessaire d'exploration dans l'arbre des possibilités est requise aussi. Dans le cadre de la planification avec espace d'actions continu, une discrétisation fine et une profondeur de recherche élevée sont deux objectifs diamétralement opposés faisant du problème du cart-pole un problème difficile à résoudre pour notre approche.

Le problème du double cart-pole

Le problème du double cart-pole est identique à celui présenté dans le chapitre précédent — voir page 76 — à l'exception de l'espace d'action A qui est maintenant continu et défini par le domaine $[-10, 10] \times [-10, 10]$.

L'expérimentation sur le problème du double cart-pole consiste à calculer la moyenne sur 1000 états initiaux et pour différentes constantes de Lipschitz $L \in \{0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.75, 1\}$ et nombres $n \in \{100, 1000, 10000, 100000\}$ d'appels au modèle génératif de la somme des récompenses sur 100

pas de temps. Les états initiaux sont tirés uniformément sur le domaine $[-2, 2] \times [-1, 1] \times [1, 5.28] \times [-1, 1] \times [-2, 2] \times [-1, 1] \times [1, 5.28] \times [-1, 1]$, les même contraintes que précédemment leur étant appliquées.

Nous pouvons observer sur le graphique 5.7 que pour les constantes de Lipschitz au dessus 0.5, les performances diminuent fortement. Pour les constantes de Lipschitz supérieures ou égales à 0.4, les performances diminuent aussi mais beaucoup plus lentement, le maximum étant atteint pour une constante de Lipschitz égale à 0.4. L'algorithme de planification lipschitzienne obtient des performances nettement inférieures à celles de l'algorithme de planification optimiste dont la moyenne de la somme des récompenses pour $n = 100$ est supérieure au maximum atteint par l'algorithme de planification lipschitzienne. Une partie de ces mauvaises performances peut être imputée au fait que la fonction récompense est fortement discontinue à cause des contraintes de distance entre les deux cart-pole.

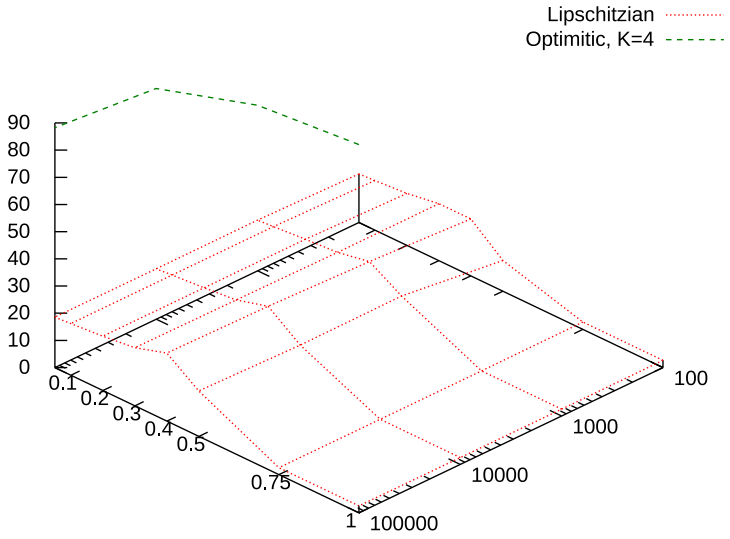


FIGURE 5.7 – Somme des récompenses pour le problème du double cart-pole.

Le problème de l'acrobot

Le problème de l'acrobot est identique à celui présenté dans le chapitre précédent — voir page 69 — à l'exception de l'espace d'action A qui est maintenant continu et défini par le domaine $[-2, 2]$.

L'expérimentation sur le problème de l'acrobot consiste à calculer la moyenne sur 1000 états initiaux et pour différentes constantes de Lipschitz

$L \in \{0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.75, 1\}$ et nombres $n \in \{100, 1000, 10000, 100000\}$ d'appels au modèle génératif de la somme des récompenses sur 100 pas de temps. Les états initiaux sont tirés uniformément sur le domaine $[1, 5.28] \times [-1, 1] \times [1, 5.28] \times [-1, 1]$.

Sur le graphique 5.8, nous pouvons observer que les performances de l'algorithme de planification lipschitzienne sont relativement proches de celles obtenues par l'algorithme de planification optimiste avec un facteur de branchement $K = 2$ par rapport aux problèmes précédents. Avec un facteur de branchement $K = 5$, l'algorithme de planification optimiste est moins bon que l'algorithme de planification lipschitzienne pour $n \leq 10000$. De plus, pour toutes les constantes de Lipschitz utilisées et contrairement aux résultats précédents, les performances de l'algorithme de planification lipschitzienne augmentent lorsque le nombre d'appels au modèle génératif augmente. Tout comme les mauvaises performances dans le cas du problème du double cart-pole pouvaient être imputées à la discontinuité de la fonction récompense, les bonnes performances rencontrées ici peuvent être dû à l'absence de discontinuité de la fonction récompense. Le maximum est atteint pour la constante de Lipschitz $L = 0.05$.

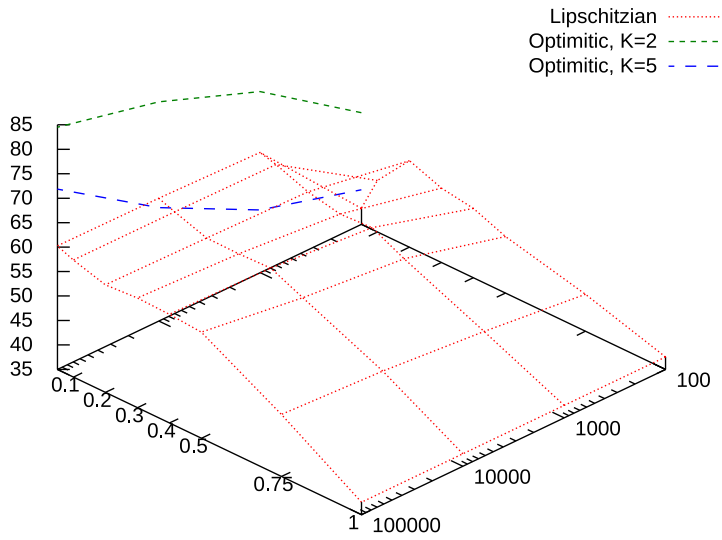


FIGURE 5.8 – Somme des récompenses pour le problème de l'acrobot.

Le problème du bateau

Le problème du bateau a pour objectif de faire traverser une rivière à un bateau — d'une berge à un ponton — en prenant en compte le courant de

la rivière — voir la figure 5.9. Les dynamiques du système — provenant de [Lazaric et al., 2007] — sont définies par les équations suivantes :

$$\begin{aligned}
p_{x,t+1} &= \min(200, \max(0, p_{x,t} + s_{t+1}\cos(\delta_{t+1}))) \\
p_{y,t+1} &= \min(200, \max(0, p_{y,t} - s_{t+1}\sin(\delta_{t+1}) - E(p_{x,t+1}))) \\
\delta_{t+1} &= \delta_t + I\Omega_{t+1} \\
\Omega_{t+1} &= \Omega_t + ((\omega_{t+1} - \Omega_t)(s_{t+1}/s_{\max})) \\
s_{t+1} &= s_t + (s_D - s_t)I \\
\omega_{t+1} &= \min(\max(p(a - \delta_t), -\pi/4), \pi/4)
\end{aligned}$$

où $p_{x,t}$ est la position en abscisse du bateau à l'instant t, $p_{y,t}$ est la position en ordonnée du bateau à l'instant t, s_t est la vitesse du bateau à l'instant t, δ_t est l'angle du bateau à l'instant t, $E(p_x)$ est l'effet du courant sur le bateau, $I = 0.1$ représente l'inertie du système, ω_t est l'angle du gouvernail à l'instant t, $s_{\max} = 2.5$ est la vitesse maximum que peut atteindre le bateau, $s_D = 1.75$ est la vitesse cible du bateau, $p = 0.9$ est un coefficient utilisé pour calculer l'angle du gouvernail dans le but d'atteindre la direction désirée a — l'action appliquée. L'effet du courant sur le bateau est défini par :

$$E(p_x) = f_c\left(\frac{p_x}{50} - \left(\frac{p_x}{100}\right)^2\right)$$

où $f_c = 1.25$ est la force du courant.

L'état du système à l'instant t est représenté par le sextuplet $(p_{x,t}, p_{y,t}, \delta_t, \Omega_t, s_t, \omega_t)$. La fonction récompense est différente de celle définie par [Lazaric et al., 2007] pour que les récompenses soient entre 0 et 1 et pour qu'elle soit plus directrice. Elle est définie pour un état $x_t = (p_{x,t}, p_{y,t}, \delta_t, \Omega_t, s_t, \omega_t)$ et une action a_t par l'équation :

$$r(x_t, a_t) = \begin{cases} 0 & \text{si } p_{x,t+1} = g^x \text{ et } |p_{y,t+1} - g_y| > g_w, \\ 1 & \text{si } p_{x,t+1} = g^x \text{ et } |p_{y,t+1} - g_y| \leq g_w, \\ \frac{\sqrt{(p_{x,t+1}-g_x)^2 + (p_{y,t+1}-g_y)^2}}{200\sqrt{2}} & \text{sinon} \end{cases}$$

où $g_x = 200$ est la position en abscisse du ponton, $g_y = 110$ est la position en ordonnée du ponton et g_w est la demi-largeur du ponton. Le facteur de dépréciation γ est de 0.95.

L'expérimentation sur le problème du bateau consiste à calculer la moyenne sur 1000 états initiaux et pour différentes constantes de Lipschitz $L \in \{0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.75, 1\}$ et nombres $n \in \{100, 100, 10000, 100000\}$ d'appels au modèle génératif de la somme des récompenses sur 200 pas de temps. Les états initiaux sont tirés uniformément sur le domaine $[0, 0] \times [0, 200] \times [0, 0] \times [-1.57, 1.57] \times [0, 0] \times [0, 0] \times [0, 0]$.

Nous pouvons observer sur le graphique 5.10 que les performances de l'algorithme de planification lipschitzienne sont équivalentes à celles obtenues par l'algorithme de planification optimiste avec un facteur de branchement

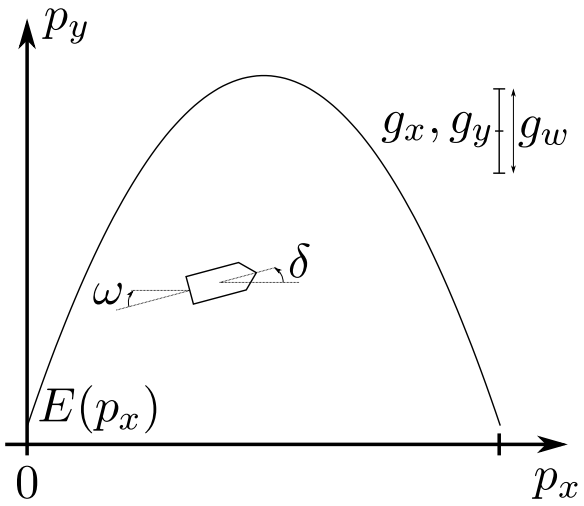


FIGURE 5.9 – Le problème du bateau.

$K = 2$. Cependant un creux de performance est présent et possède un minimum pour la constante de Lipschitz $L = 0.3$ lorsque le nombre d'appels au modèle génératif est égale à 100000. Ce creux s'explique par la nature très directive de la fonction récompense faisant qu'une simple discrétisation de l'espace d'action associé à la première prise de décision permet d'obtenir de bonne performance. Ceci est vérifié par les bonnes performances obtenues par des constantes de Lipschitz supérieure ou égale à 0.4 qui privilégient l'exploration en largeur. À l'inverse, lorsque la constante de Lipschitz est très faible, la fonction récompense amplifie la recherche en profondeur. Ainsi pour une constante de Lipschitz $L = 0.05$, la profondeur explorée est suffisante pour trouver des trajectoires viables augmentant les performances obtenues.

Le problème de la lévitation magnétique d'une boule en acier

Le problème de la lévitation magnétique d'une boule en acier consiste à positionner une boule en acier à l'aide d'un champs électromagnétique selon une trajectoire verticale en contrôlant le voltage appliqué à la bobine — voir la figure 5.11. Les dynamiques du système — provenant de [Hafner and Riedmiller, 2011] — sont définies par les équations suivantes :

$$\begin{aligned} \dot{p} &= v \\ \dot{v} &= g - \frac{\xi I^2}{2M(x_\infty + p)^2} \\ \dot{I} &= \frac{I(\xi v - R(x_\infty + p)^2)}{\xi(x_\infty + p) + L_\infty(x_\infty + p)^2} + a\frac{x_\infty + p}{\xi + L_\infty(x_\infty + p)} \end{aligned}$$

avec $p \in [0, 0.013]$ la position verticale de la boule en acier, v sa vitesse, I le courant dans la bobine, $a \in [-60, 60]$ le voltage appliqué à la bobine, $g = 8.91$ la pesanteur, $M = 0.8$ la masse de la boule d'acier, $R = 11.68$

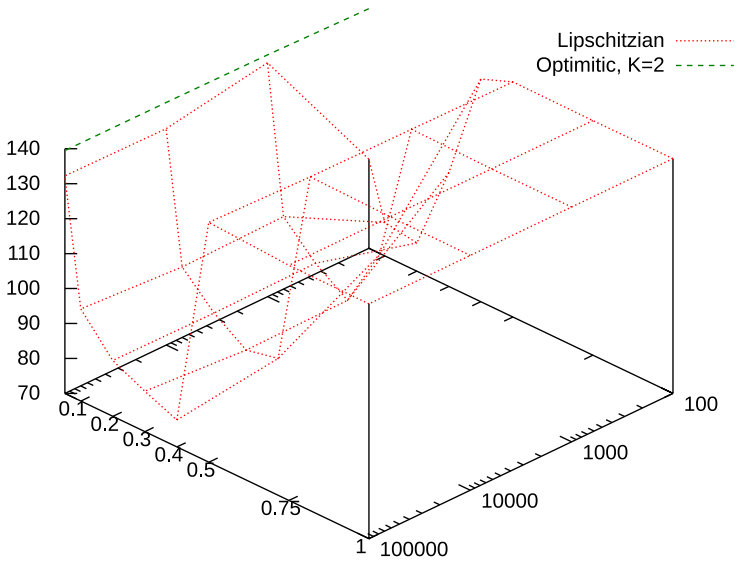


FIGURE 5.10 – Somme des récompenses pour le problème du bateau.

la résistance électrique, $x_\infty = 0.0007$, $L_\infty = 0.8052$ et $\xi = 0.001599$ les paramètres de la bobine.

L'état du système à l'instant t est représenté par le triplet (p_t, v_t, I_t). La fonction récompense est définie pour un état $x_t = (p_t, v_t, I_t)$ et une action a_t par l'équation

$$r(x_t, a_t) = 1 - \frac{|p_{t+1} - p_g|}{0.013}$$

avec p_g la position à atteindre. Le facteur de dépréciation γ est de 0.9 et le pas de temps de 0.004. Les dynamiques du système sont intégrées en utilisant la méthode d'intégration Runge Kutta 4 avec 2 étapes intermédiaires.

L'expérimentation sur le problème de la lévitation magnétique d'une boule en acier consiste à calculer la moyenne de la somme des récompenses obtenues pour 1000 objectifs de position à atteindre par la boule en acier pour différentes constantes de Lipschitz $L \in \{0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.75, 1\}$ et nombres $n \in \{100, 1000, 10000, 100000\}$ d'appels au modèle génératif. L'objectif de position change tous les 80 pas de temps. Les objectifs de position suivants ne sont pas connus à l'avance — c'est un paramètre du modèle génératif changé tout les 80 pas de temps. L'état initial est obtenu en appliquant l'action correspondant à un courant de 15V pendant 125 pas de temps en partant de l'état $(0.013, 0, 0)$ correspondant à l'état de repos de la boule en acier. Le premier objectif de position est alors effectif pour 80 pas de temps. Une fois ces 80 pas de temps écoulés, l'objectif de position est changé

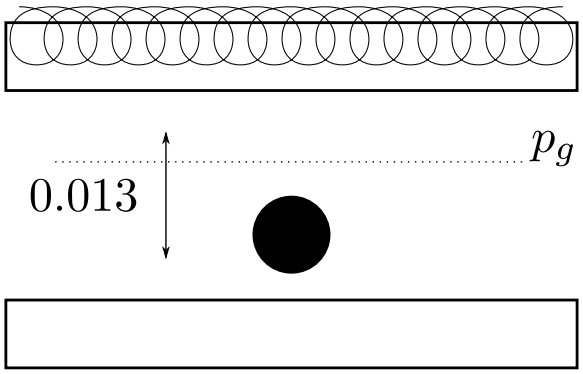


FIGURE 5.11 – Le problème de la lévitation magnétique d'une boule en acier.

en conservant l'état courant du système — autrement dit, nous ne repartons pas de l'état initial. Les objectifs de position sont tirés uniformément sur le domaine $[0, 0.013]$.

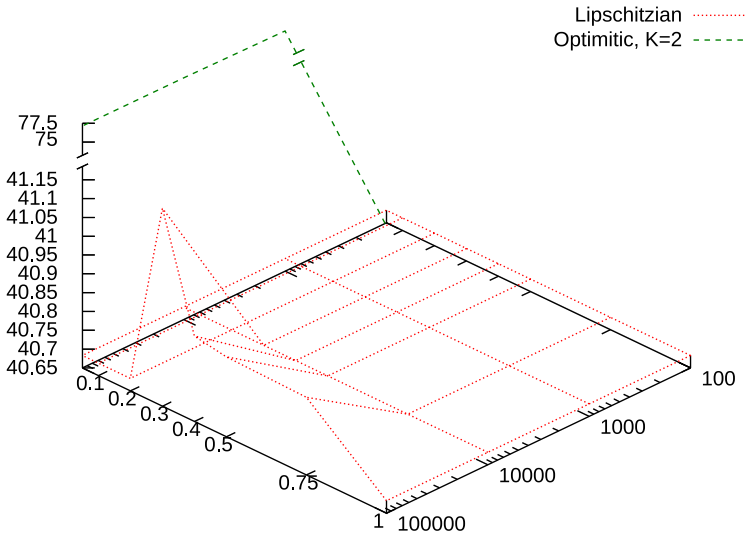


FIGURE 5.12 – Somme des récompenses pour le problème de la lévitation magnétique d'une boule en acier.

Nous pouvons observer sur le graphique 5.12 que les performances de l'algorithme de planification Lipschitzienne sont en retrait par rapport à celles obtenues par l'algorithme de planification optimiste avec un facteur de branchement $K = 2$. De plus, les performances ne varient que très peu en fonction de la constante de Lipschitz. Ce comportement est dû aux dy-

namiques du système qui possède une fonction de transition non linéaire, une grande instabilité et des mouvements très rapide. De fait, la vraie constante de Lipschitz pour ce problème est vraisemblablement élevée ce qui dans le cadre de notre algorithme provoquerait une exploration uniquement en largeur ce qui n'est pas compatible avec la nature instable des dynamiques demandant une exploration profonde.

5.1.3 Discussion

Comme nous avons pu l'observer au cours des différentes expérimentations, les résultats pour l'algorithme de planification lipschitzienne ne sont pas probants. La raison principale se situe dans l'hypothèse de départ qui demande que la fonction transition ainsi que la fonction récompense soient lipschitziennes. Cependant cette hypothèse est trop forte pour obtenir des résultats convainquant.

De plus l'estimation de la constante de Lipschitz L à fournir à l'algorithme est une tâche aussi complexe que la résolution du problème lui même. En supposant que nous connaissions la constante de Lipschitz spécifique au problème, celle-ci serait une borne supérieure sur les constantes $L_{f_x}, L_{f_a}, L_{r_x}, L_{x_a}$ produisant une borne supérieure pas assez fine et altérant ainsi les performances de l'algorithme.

Il est à noter que l'utilisation même d'une constante de Lipschitz produit une borne supérieure dont la porté est globale. Il serait plus opportun si la borne supérieure étant plus en phase avec le paysage local de la fonction de transition et de la fonction récompense.

Nous pouvons prendre l'exemple du problème du cart-pole où deux phases distinctes sont visibles.

La première consiste à augmenter la vitesse angulaire du mat pour le mettre en position verticale demandant ainsi une recherche plus en profondeur pour obtenir le mouvement de balancier. Cette recherche en profondeur est favorisée par une constante de Lipschitz faible incitant à la prolongation des séquences d'actions plutôt qu'à la trisection de sous-espaces.

Dans la seconde phase, le mat doit être maintenu en équilibre ce qui demande des actions plus précises et donc une discrétisation plus fine dans les profondeurs les plus faibles. Cette exploration en largeur est obtenue par une constante de Lipschitz élevée favorisant la trisection des sous-espaces plutôt que la prolongation des séquences d'actions.

Par cet exemple, on peut ainsi voir qu'il est problématique de choisir une constante de Lipschitz car il peut être intéressant pour un même problème mais suivant l'état courant de l'adapter ce qui comme l'estimation n'est pas chose facile.

Ainsi la constante de Lipschitz est un paramètre déterminant la nature de l'exploration de l'arbre des possibilités. Cela n'est pas sans rappeler [Maes et al., 2011] qui propose d'optimiser les paramètres d'une politique de

recherche dans l'arbre des possibilités et ce en utilisant le temps disponible hors ligne. Ces paramètres sont optimisés sur un échantillon représentatif des états atteignables du problème courant puis ils sont utilisés pour contrôler le système défini par le problème en ligne. Cette approche consistant à utiliser le temps hors ligne pour optimiser un paramètre dépendant du problème courant peut être une voie à explorer en vue de trouver une constante de Lipschitz adaptée à notre algorithme de planification lipschitzienne.

En dehors de la difficulté de déterminer la constante de Lipschitz d'un problème, les ressources computationnelles nécessaires à l'exécution de l'algorithme ne sont pas négligeables. Ceci est dû d'une part aux différents calculs de bornes effectués pour choisir quel type de discrétisation appliquer et d'autre part à la complexité de la structure de données requise pour stocker les sous-ensembles de sous-espaces ainsi que les valeurs associées. En effet, si les ressources computationnelles sont exprimées en terme de limite de temps entre deux prises de décisions, il devient alors nécessaire de minimiser le temps propre à l'algorithme vis-à-vis du temps utilisé par le modèle génératif permettant ainsi de maximiser le nombre d'appels à celui-ci.

De plus et contrairement aux algorithmes de planification uniforme et optimiste, la réutilisation entres deux prises de décisions d'un sous-ensemble des séquences d'actions explorées est complexe.

Dans le cas des algorithmes de planification uniforme et optimiste, Il convient de garder le sous-arbre correspondant à l'action optimale retournée précédemment dans le cas où le nouvel état du système issu de cette action est identique à l'état présent en racine du sous-arbre.

Dans le cas de l'algorithme de planification lipschitzienne, il faut garder les sous-ensemble A_i dont l'action a_{c0}^i est identique à celle retournée par l'algorithme et ce seulement si le nouvel état du système est identique à ceux étendus par l'algorithme. Il convient donc de lier ensemble les sous-ensembles A_i possédant la même première action a_{c0}^i compliquant encore la structure de données. Cependant l'union des sous-espaces b_0^i des sous-ensemble A_i récupérés de l'utilisation précédente de l'algorithme de planification lipschitzienne peut ne pas être égale à A. Il est donc nécessaire de calculer le sous-espace de A non couvert par les sous-espaces b_0^i. Préserver les calculs effectués lors de la prise de décision précédente est donc un choix viable si et seulement si un appel au modèle génératif est couteux en terme de ressources computationnelles.

Dans le chapitre 3, nous avons présenté l'algorithme HOOT — voir page 39 — combinant l'algorithme UCT et l'algorithme HOO. En partant du même principe, nous pouvons construire un nouvel algorithme tirant parti des régularités présentes dans le problème pour discrétiser l'espace d'action mais en empruntant une approche plus intuitive que théorique que celle utilisée dans cette section et essayer ainsi d'obtenir de meilleures performances.

5.2 Planification Séquentielle

Dans cette section, nous allons présenter un nouvel algorithme de planification avec espace d'action continu pouvant être utilisé conjointement avec l'algorithme Dividing Rectangles (DIRECT) ou avec l'algorithme Simultaneous Optimistic Optimization (SOO). Cet algorithme de planification séquentielle se propose de reprendre le principe utilisé dans l'algorithme HOOT mais adapté au cas déterministe. Contrairement à l'algorithme de planification lipschitzienne, l'algorithme présenté dans ce chapitre ne repose pas sur des fondations théoriques mais sur une approche intuitive.

Nous allons dans un premier temps présenter les algorithmes DIRECT et SOO ainsi que l'approche envisagée et l'algorithme de planification en découlant. Dans un second temps, nous présentons les résultats expérimentaux obtenus pour chacune des deux variantes.

5.2.1 Description

Les algorithmes DIRECT et SOO sont des algorithmes d'optimisation globale déterministe utilisant la régularité de la fonction à optimiser pour orienter la discrétisation de l'espace des paramètres. Nous allons en premier lieu présenter les deux algorithmes puis en second lieu la définition formelle de l'algorithme de planification séquentielle.

L'algorithme DIRECT

L'algorithme Dividing Rectanles (DIRECT) de [Jones et al., 1993] est un algorithme d'optimisation globale déterministe utilisant une approche lipschitzienne mais sans requérir la constante de Lipschitz. Pour ce faire, cette approche envisage toutes les valeurs possibles pour la constante de Lipschitz rendant ainsi sa connaissance exacte inutile. Contrairement à une constante de Lipschitz couvrant complètement l'espace des paramètres, les constantes de Lipschitz envisagées sont dépendantes du paysage locale de la fonction optimisée évitant ainsi d'utiliser une estimation haute de la constante de Lipschitz qui encouragerait une recherche globale freinant ainsi la convergence vers l'optimum. Il est à noter que la version présentée par [Jones et al., 1993] permet de trouver le minimum d'une fonction, nous présentons ici celle permettant de trouver le maximum d'une fonction.

L'algorithme DIRECT sélectionne un sous-espace de l'espace des paramètres pour être discrétisé si le point associé, ayant pour abscisse la taille du sous-espace et pour ordonnée la valeur retournée par la fonction à optimiser avec ce point comme paramètre, appartient à l'enveloppe convexe supérieure droite. Les sous-espaces sélectionnés sont discrétisés en les trisectant. Cela permet de conserver au centre des sous-espaces discrétisés les points évalués et ainsi économiser les évaluations. Cette méthode a d'ailleurs

été reprise dans l'algorithme de planification lipschitzienne lorsqu'un sous-espace est discrétisé.

De manière plus formelle, nous considérons la fonction à optimiser $f : A \to \mathbb{R}$ avec A l'espace des paramètres de dimension 1. Soit $\mathcal{A}_i \subset A$ les sous-espaces déjà discrétisés avec $i \in \{1, \ldots, n\}$, ayant pour centre a_{ci} et pour diamètre $\delta_i = \max_{a \in \mathcal{A}_i} \|a_{ci} - a\|$. Soit $\epsilon > 0$ une constante positive et $f_{\max} = \max_{i \in \{1,\ldots,n\}} f(a_{ci})$ la valeur maximum rencontrée pendant les itérations précédentes. Un sous-espace $\mathcal{A}_j$ avec $j \in \{1, \ldots, n\}$ est dit potentiellement optimal si il existe une pente $\tilde{L} > 0$ telle que pour tout $i \in \{1, \ldots, n\}$

$$f(c_j) + \tilde{L}\delta_j \geq f(c_i) + \tilde{L}\delta_i$$

et

$$f(c_j) + \tilde{L}\delta_j \geq f_{\max} - \epsilon|f_{\max}|.$$

Les sous-espaces potentiellement optimaux sont ensuite trisectés et le centre des sous-espaces ainsi créés est évalué par la fonction à optimiser f. Ce processus est itéré jusqu'à ce que les ressources computationnelles disponibles sont épuisées — le nombre d'appels à la fonction f par exemple — ou qu'une précision particulière a été atteinte — le diamètre du plus petit sous-espace par exemple. Si l'espace des paramètres A possède plus d'une dimension, son côté le plus large est trisecté. Il est à noter que l'espace des paramètres A est mis à l'échelle pour prendre la forme d'un hypercube. L'algorithme DIRECT est explicité par l'algorithme 5.2.

Algorithme 5.2 Dividing Rectangles

$\mathcal{A}_1 \leftarrow A$
$n \leftarrow 1$
$f_{\max} \leftarrow f(a_{c1})$
$a_{ci_{\max}} \leftarrow 1$
while Une condition d'arrêt n'est pas vraie **do**
  Identifier l'ensemble $I \subseteq \{\mathcal{A}_i | 1 \leq i \leq n\}$ des sous-espaces potentiellement optimaux
  for all $\mathcal{A} \in I$ **do**
    Trisecter le sous-espace $\mathcal{A}$ suivant son côté le plus long
    Évaluer $f(a_{cn+1})$ et $f(a_{cn+2})$
    Remettre à jour $f_{\max}$ et $a_{ci_{\max}}$
    $n \leftarrow n + 2$
  end for
end while
return $a_{ci_{\max}}$

L'algorithme Simultaneous Optimistic Optimization

L'algorithme Simultaneous Optimistic Optimization (SOO) de [Munos, 2011] est un algorithme d'optimisation globale qui suppose l'existence d'une régularité sur la fonction à optimiser mais ne requière pas sa connaissance. Le principe est donc le même que l'algorithme DIRECT mais pour une classe de régularités plus large. L'algorithme SOO discrétise l'espace des paramètres de la fonction à optimiser de manière hiérarchique où chaque nœud et feuille de l'arbre est associé à un sous-espace de l'espace des paramètres. À chaque itération, l'algorithme SOO sélectionne tout au plus une feuille à chaque profondeur de l'arbre pour discrétisation. Nous supposerons que comme l'algorithme DIRECT, un sous-espace est discrétisé en le trisectant. La profondeur de l'arbre est limitée par une fonction $h_{\max}(t)$ où t est le nombre de feuilles dont le sous-espace associé a été discrétisé. Cette fonction permet de répartir l'effort de recherche entre une discrétisation plus fine et une discrétisation plus uniforme.

Une feuille est sélectionnée et son sous-espace associé discrétisé si sa valeur — le retour de la fonction à optimiser en son centre — est plus grande que la valeur de toutes les autres feuilles ayant la même profondeur et plus grande que la valeur maximum parmi les feuilles de profondeur plus faible discrétisées pendant l'itération courante.

De manière plus formelle, nous définissons $\mathcal{L}_t$ comme l'ensemble des feuilles (h, i) à l'instant t de profondeur h et d'index i appartenant à l'arbre $\mathcal{T}_t$ de profondeur d_t. Nous définissons aussi $x_{h,i}$ le point d'évaluation de la fonction à optimiser f associé à la feuille — ou au nœud — (h, i). Le sous-espace $\mathcal{A}_i$ associé à une feuille (h, i) sélectionnée est trisecté et les 3 nouvelles feuilles $(h+1, 3i-1)$, $(h+1, 3i)$ et $(h+1, 3i+1)$ sont ajoutées à l'arbre $\mathcal{T}_t$. L'algorithme SOO est défini par l'algorithme 5.3.

L'algorithme de Planification Séquentielle

Nous avons vu au chapitre 3 à la page 39 l'algorithme HOOT combinant l'algorithme UCT et l'algorithme HOO. L'idée principale de cet algorithme était d'instancier l'algorithme HOO à chaque nœud d'un arbre développé selon l'algorithme UCT pour discrétiser l'espace d'action continu en actions applicables à l'état associé. Cette approche peut être reprise dans un cadre déterministe mais en apportant quelques modifications.

Étant donné que, dans le cas déterministe, un état ne possède qu'un seul état successeur possible pour une action donnée, discrétiser l'espace d'action impacte directement sur le nombre d'états successeurs et donc sur le facteur de branchement d'un arbre des possibilités si nous omettons de regrouper les états successeurs identiques ensembles. Cependant même avec regroupement, le nombre d'états successeurs peut être élevé. Ce problème est réglé par l'algorithme HOOT en regroupant les états se trouvant dans le

Algorithme 5.3 Simultaneous Optimistic Optimization

$t \leftarrow 1$
$\mathcal{T}_t \leftarrow (0, 1)$
$\mathcal{L}_t \leftarrow (0, 1)$
while Une condition d'arrêt n'est pas vraie **do**
  $v_{\max} \leftarrow -\infty$
  for $h \leftarrow 0$ **to** $\min(d_t, h_{\max}(t))$ **do**
    if $\{(h, i) | (h, i) \in \mathcal{L}_t\} \neq \emptyset$ **then**
      $(h, i) \leftarrow \arg\max_{(h,j) \in \mathcal{L}_t} f(x_{h,j})$
      if $f(x_{h,i}) \geq v_{\max}$ **then**
        Trisecter le sous-espace $i\mathcal{A}_i$
        Ajouter les feuilles $(h + 1, 3i - 1)$, $(h + 1, 3i)$ et $(h + 1, 3i + 1)$ à l'arbre $\mathcal{T}_t$
        $v_{\max} \leftarrow f(x_{h,i})$
        $t \leftarrow t + 1$
      end if
    end if
  end for
end while
return $\arg\max_{(h,i) \in \mathcal{L}_t} f(x_{h,i})$

même sous-espace issus d'une discrétisation prédéfinie de l'espace des états. L'algorithme que nous proposons dans ce chapitre prend le chemin contraire et s'inspire de l'algorithme OLOP présenté au chapitre 3 à la page 35.

Ainsi, plutôt que de regrouper les états successeurs, nous allons les ignorer et observer seulement des séquences d'actions pour planifier en boucle ouverte comme l'algorithme OLOP. Le résultat se résume à une séquence d'instances d'un algorithme d'optimisation sur l'espace d'action continu. Chaque instance nous fournit à chaque itération une action participant à la construction d'une séquence d'actions. Une fois les récompenses obtenues, les différentes instances sont mises à jour. La longueur de la séquence d'instances $H \geq 1$ est un paramètre de l'algorithme qui est décrit par l'algorithme 5.4. Cette longueur est fixée car la changer en cours de planification introduirait un biais vers les récompenses obtenues plus tardivement dans les algorithmes d'optimisation globale ce qui n'est pas souhaitable. Comme l'algorithme AMS présenté au chapitre 3 à la page 29, nous pouvons considérer que l'algorithme de planification séquentielle sera mieux adapté aux problèmes modélisés par un MDP à horizon fini H. Dans le cas présent, nous allons cependant l'utiliser dans le cadre des MDPs à horizon infini.

Une autre façon d'appréhender l'algorithme de planification séquentielle est en prenant comme base l'algorithme de recherche aléatoire comme décrit à la page 60. Dans cet algorithme, les séquences d'actions simulées grâce au

Algorithme 5.4 Planification Séquentielle

Initialiser les H instances de l'algorithme d'optimisation globale
Initialiser $\tilde{a}^*$ comme étant le centre de l'espace d'action A
$\tilde{q}^* \leftarrow \sum_{t=0}^{H-1} \gamma^t r_t(\mathbf{a}^*)$ avec $\mathbf{a}^* = \{a^*, \ldots, a^*\} \in A^H$
while Une condition d'arrêt n'est pas vraie **do**
  $\tilde{q} \leftarrow 0$
  for $t \leftarrow 0$ **to** $H-1$ **do**
    Demander une action a_t à l'instance t
    $\tilde{q}_t \leftarrow r_t(x_t, a_t)$
  end for
  for $t \leftarrow H-1$ **to** 0 **do**
    $\tilde{q} \leftarrow \tilde{q}_t + \gamma\tilde{q}$
    Mettre à jour l'instance t
  end for
  if $\tilde{q} > \tilde{q}^*$ **then**
    $\tilde{q}^* \leftarrow \tilde{q}$
    $\tilde{a}^* \leftarrow a_0$
  end if
end while
return $\tilde{a}^*$

modèle génératif sont formées par une séquence de tirages uniformes discrets sur l'espace d'action discret. L'extension dans le cas d'un espace d'action continu revient ainsi à effectuer un tirage uniforme sur l'espace d'action continu. Comme nous l'avons vu dans le chapitre précédent, l'algorithme de recherche aléatoire est robuste à l'augmentation du facteur de branchement, encourageant d'autant plus son extension dans le cas d'un espace d'action continu. Cependant il serait opportun de biaiser la distribution utilisée pour tirer plus souvent des actions provenant de sous-espaces prometteurs de l'espace d'action. Ainsi dans l'algorithme de planification séquentielle, nous remplaçons le tirage uniforme sur l'espace d'action par une instance d'un algorithme d'optimisation globale qui fournira une action appartenant au sous-espace le plus prometteur.

Le principal défaut de cette approche est d'avoir dès le départ une longueur fixe pour les séquences d'actions. Il serait plus avantageux dans le cas où les ressources computationnelles disponibles ne sont pas connues à l'avance de voir cette longueur évoluer au cours du temps. Cependant comme nous l'avons dit précédemment, augmenter la longueur des séquence d'actions après un certain nombre d'itérations provoquerait un biais vers les séquences d'actions simulées après cette augmentation. En effet, la somme des récompenses dépréciées maximale atteignable augmentant, entres deux séquences d'actions comparables, celle étant la plus longue serait avantagée. Il convient donc de prendre en compte ce biais si nous souhaitons faire évoluer

la longueur des séquences d'actions au cours de la planification.

Une solution simple est de faire évoluer la longueur des séquences d'actions au cours du temps en fonction du nombre d'appels au modèle génératif déjà effectués comme c'est le cas, par exemple, pour la recherche aléatoire. Cependant, pour prendre en compte le biais, nous devrions relancer l'algorithme de planification séquentielle avec cette nouvelle longueur.

Dans le cas où un appel au modèle génératif est extrêmement couteux, nous pouvons envisager de conserver un historique des séquences d'actions simulées en vu d'économiser les appels au modèle génératif et ceci en se basant uniquement sur les actions effectuées de façon à ne pas comparer les états eux-mêmes. Cependant, ceci restreint à l'usage d'algorithme d'optimisation globale dont les points d'évaluations ne sont pas tirés aléatoirement ce qui dans le cas contraire réduirait l'efficacité du système d'historique.

Une autre solution serait de faire évoluer la longueur des séquences d'actions en fonction d'indicateurs provenant de l'algorithme d'optimisation globale utilisé. Dans le cas de l'algorithme DIRECT, un indicateur pourrait être la taille du plus petit sous-espace discrétisé. Dans le cas l'algorithme SOO, cet indicateur pourrait être dépendant de la fonction $h_{\max}(t)$ et/ou de la profondeur atteinte.

Une utilisation annexe des algorithmes de planification avec espace d'action continu est dans le cadre des problèmes avec pas de temps continu comme décrit dans [Neumann et al., 2007]. Une solution simple est d'augmenter l'espace d'action continu avec la durée du pas de temps. Ainsi la dimension supplémentaire ajoutée à l'espace d'action sera discrétisée comme les autres dimensions formant l'action par un algorithme de planification avec espace d'action continu comme ceux présentés dans ce chapitre. Le temps continu peut être aussi défini comme une durée aléatoire de transition d'un état à un autre et est modélisé par un processus de décision semi-markovien (SMDP) — [Bradtke and Duff, 1994],[Ghavamzadeh and Mahadevan, 2001]. Cependant, la nature stochastique de cette formulation ne nous permet pas d'appliquer les algorithmes présentés dans ce chapitre.

5.2.2 Expérimentations

Les expérimentations suivantes sont conduites sur l'algorithme de planification séquentielle lorsqu'il utilise l'algorithme DIRECT ou l'algorithme SOO avec $h_{\max}(t) = \lfloor \sqrt{t} \rfloor$ et $h_{\max}(t) = \lfloor \sqrt[3]{t} \rfloor$. L'algorithme de recherche aléatoire est aussi inclus dans les expérimentations et est adapté au cas continu en tirant uniformément sur l'espace d'action, les actions appliquées le long d'une séquence de longueur fixée — contrairement au chapitre précédent où la longueur des séquences était fonction du nombre d'appels déjà effectués au modèle génératif.

Sur chacun des graphiques suivants, l'axe des abscisses correspond à la longueur des séquences H et l'axe des ordonnées à la moyenne de la somme

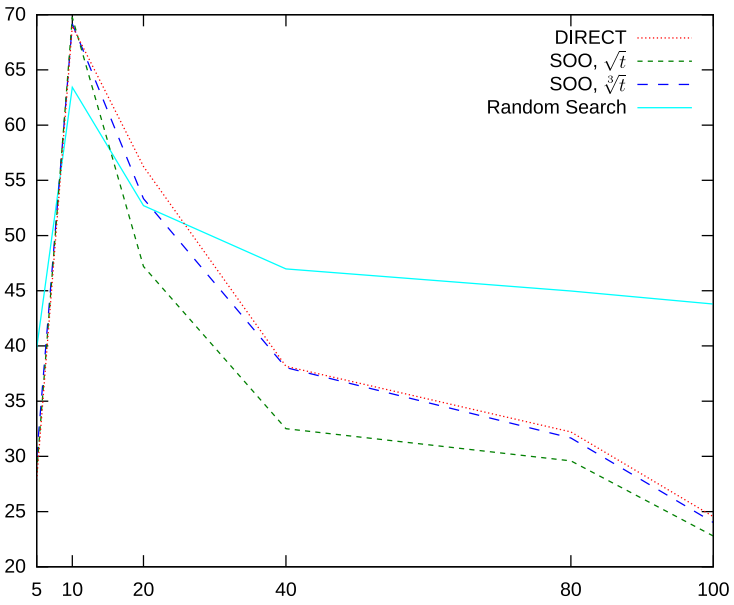


FIGURE 5.13 – Somme des récompenses pour le problème du cart-pole avec $n = 100$ appels au modèle génératif.

des récompenses obtenue.

Le problème du cart-pole

Le problème du cart-pole est identique à celui présenté dans le chapitre précédent — voir page 65 — à l'exception de l'espace d'action A qui est continu et défini par le domaine $[-10, 10]$.

L'expérimentation sur le problème du cart-pole consiste à calculer la moyenne sur 1000 états initiaux et pour différentes longueurs de séquences $H \in \{5, 10, 20, 40, 80, 100\}$ et différents nombres $n \in \{100, 1000, 10000\}$ d'appels au modèle génératif de la somme des récompenses sur 100 pas de temps. Les états initiaux sont tirés uniformément sur le domaine $[-2, 2] \times [-5, 5] \times [1, 5.28] \times [-1, 1]$.

Nous pouvons observer sur le graphique 5.13 représentant la moyenne de la somme des récompenses obtenues pour $n = 100$ appels au modèle génératif que pour $H = 10$, l'algorithme de planification séquentielle utilisant l'algorithme SOO avec $h_{\max}(t) = \lfloor \sqrt{t} \rfloor$ obtient la meilleure performance, celle-ci étant supérieure à l'algorithme de recherche aléatoire. Nous pouvons noter cependant que lorsque $H \geq 20$, les performances des variantes de l'algorithme de planification séquentielle baissent. Ce comportement est imputable à la discrétisation trop faible de l'espace d'action continu dû à l'augmentation de la longueur des séquences d'actions diminuant de même la quantité de ressources computationnelles utilisées par chaque instance des

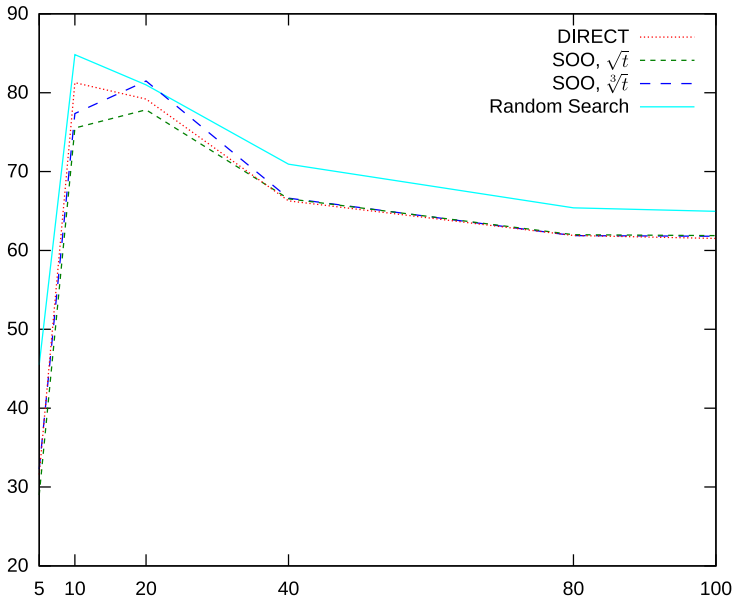


FIGURE 5.14 – Somme des récompenses pour le problème du cart-pole avec $n = 1000$ appels au modèle génératif.

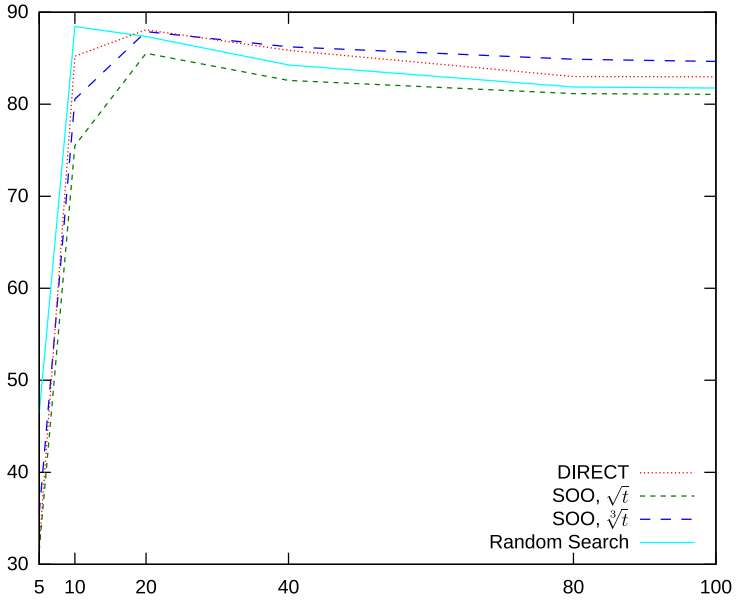


FIGURE 5.15 – Somme des récompenses pour le problème du cart-pole avec $n = 10000$ appels au modèle génératif.

algorithmes d'optimisation globale. Dans cette partie — $H \geq 20$ —, l'algorithme de planification séquentielle utilisant l'algorithme DIRECT obtient les meilleurs performances mais reste en retrait par rapport à l'algorithme de recherche aléatoire.

Le graphique 5.14 représente la moyenne de la somme des récompenses obtenues pour $n = 1000$ appels au modèle génératif. Nous pouvons observer que l'algorithme de recherche aléatoire obtient la meilleure performance pour $H = 10$. L'algorithme planification séquentielle utilisant l'algorithme SOO avec $h_{\max}(t) = \sqrt[3]{t}$ obtient sa meilleure performance pour $H = 20$ suivi de près par l'algorithme de recherche aléatoire ainsi que par l'algorithme de planification séquentielle utilisant l'algorithme DIRECT pour $H = 10$. De nouveau, les performances baissent lorsque la longueur des séquences augmente mais de manière moins marquée qu'auparavant.

Le graphique 5.15 représente la moyenne de la somme des récompenses obtenues pour $n = 10000$ appels au modèle génératif. Nous pouvons observer que l'algorithme de planification séquentielle utilisant l'algorithme DIRECT obtient sa meilleure performance pour $H = 20$ mais celle-ci est inférieure à la performance obtenue par l'algorithme de recherche aléatoire pour $H = 10$. Le nombre d'appels au modèle génératif augmentant, la diminution des performances lors de l'augmentation de la longueur des séquences est très faible permettant même à deux variantes sur trois de l'algorithme de planification séquentielle d'obtenir de meilleures performances que l'algorithme de recherche aléatoire lorsque $H \geq 20$.

Globalement, les performances des variantes de l'algorithme de planification séquentielle sont proches de celles l'algorithme de recherche aléatoire, leur principal avantage étant qu'elles définissent une politique déterministe contrairement à celle générée par l'algorithme de recherche aléatoire qui est stochastique.

Le problème du double cart-pole

Le problème du double cart-pole est identique à celui présenté dans le chapitre précédent — voir page 76 — à l'exception de l'espace d'action A qui est continu et défini par le domaine $[-10, 10] \times [-10, 10]$.

L'expérimentation sur le problème du cart-pole consiste à calculer la moyenne sur 1000 états initiaux et pour différentes longueurs de séquences $H \in \{5, 10, 20, 40, 80, 100\}$ et différents nombres $n \in \{100, 1000, 10000\}$ d'appels au modèle génératif de la somme des récompenses sur 100 pas de temps. Les états initiaux sont tirés uniformément sur le domaine $[-2, 2] \times [-1, 1] \times [1, 5.28] \times [-1, 1] \times [-2, 2] \times [-1, 1] \times [1, 5.28] \times [-1, 1]$, les même contraintes que précédemment leur étant appliquées.

Nous pouvons observer sur le graphique 5.16, représentant la moyenne de la somme des récompenses obtenues pour $n = 100$ appels au modèle génératif, que l'algorithme de planification séquentielle utilisant l'algorithme

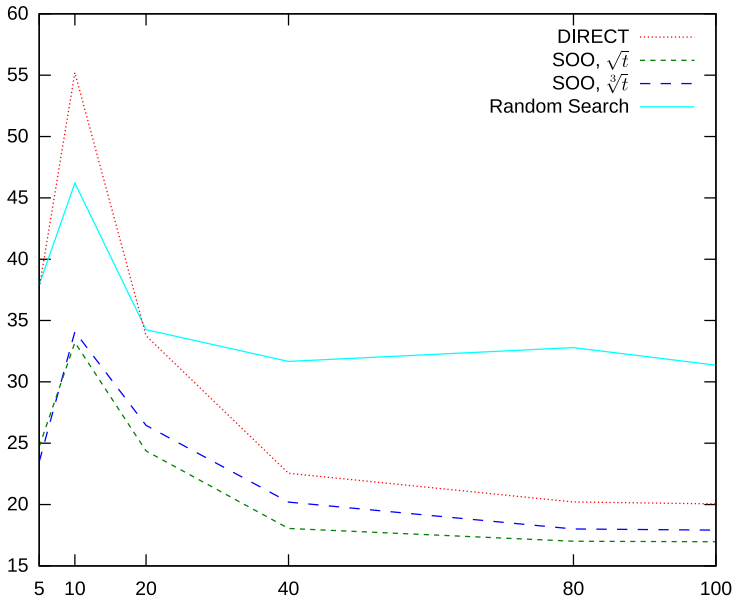


FIGURE 5.16 – Somme des récompenses pour le problème du double cart-pole avec $n = 100$ appels au modèle génératif.

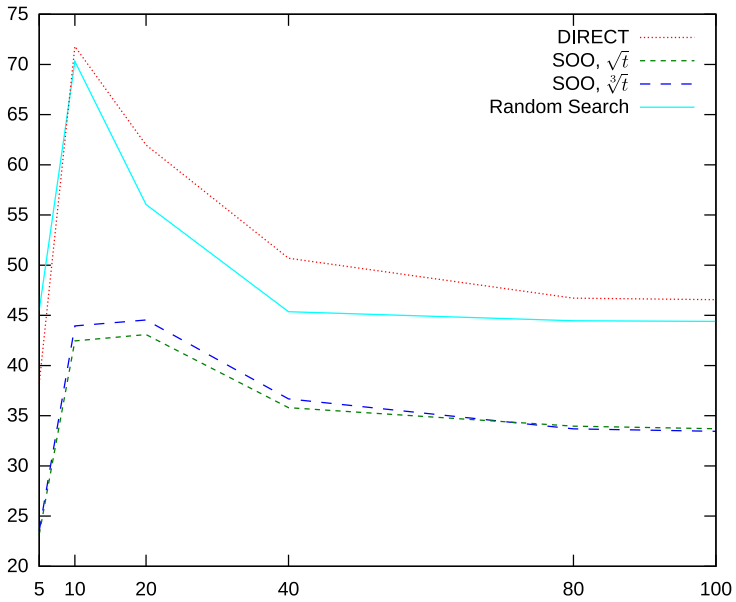


FIGURE 5.17 – Somme des récompenses pour le problème du double cart-pole avec $n = 1000$ appels au modèle génératif.

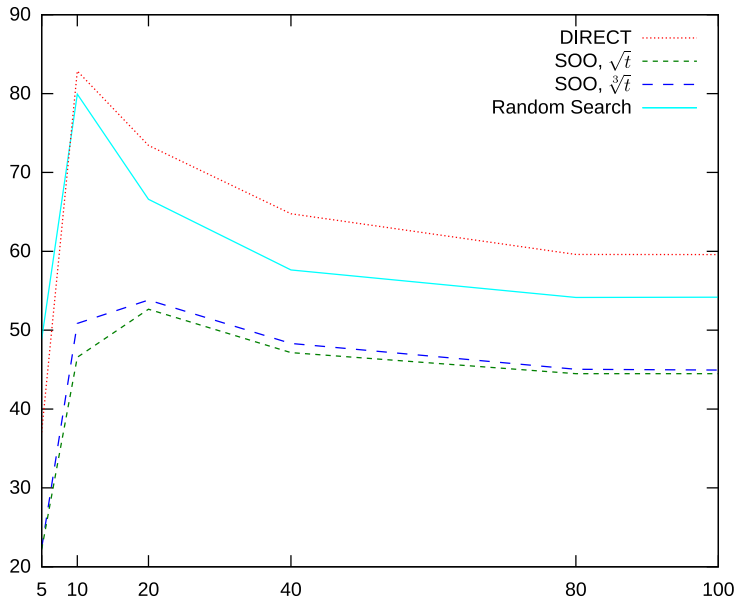


FIGURE 5.18 – Somme des récompenses pour le problème du double cart-pole avec $n = 10000$ appels au modèle génératif.

DIRECT obtient les meilleures performances pour $H = 10$. Pour $H \geq 20$, l'algorithme de recherche aléatoire obtient de meilleures performances que les trois variantes de l'algorithme de planification séquentielles. Les deux variantes utilisant l'algorithme SOO sont en retrait et obtiennent globalement des performances inférieures à l'algorithme de recherche aléatoire. La forte diminution des performances de la variante utilisant l'algorithme DIRECT est dû à l'augmentation de la longueur des séquences diminuant de fait la part de ressources computationnelles distribuées à chacune de ses instances d'autant plus que l'espace d'action est de dimension 2.

Le graphique 5.17 représente la moyenne de la somme des récompenses obtenues pour $n = 1000$ appels au modèle génératif. L'algorithme de planification séquentielle utilisant l'algorithme DIRECT obtient de nouveau les meilleures performances, dominant presque entièrement l'algorithme de recherche aléatoire lui même dominant les deux variantes de l'algorithme de planification séquentielle utilisant l'algorithme SOO.

Le graphique 5.18 représente la moyenne de la somme des récompenses obtenues pour $n = 10000$ appels au modèle génératif. De nouveau l'algorithme de planification séquentielle utilisant l'algorithme DIRECT obtient les meilleures performances avec un maximum atteint pour $H = 10$.

Un facteur peut rentrer en compte pour expliquer la différence de performance entre la variante avec l'algorithme DIRECT et les variantes avec l'algorithme SOO, c'est le nombre de dimensions de l'espace d'action. Celui-

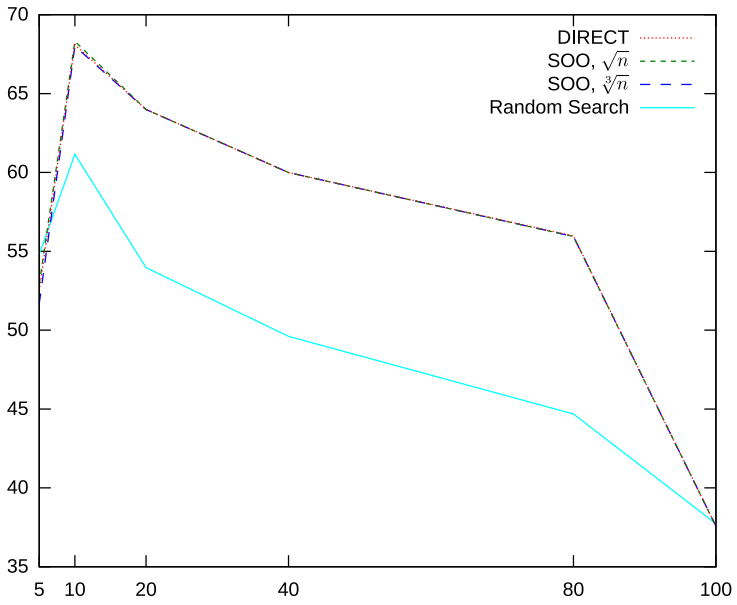


FIGURE 5.19 – Somme des récompenses pour le problème de l'acrobot avec $n = 100$ appels au modèle génératif.

ci étant de 2 et les deux algorithmes d'optimisation globale n'ayant pas la même stratégie de discrétisation, son augmentation peut mettre en exergue leur différence. De plus, ce nombre de dimensions accru peut aussi expliquer l'écart de performance en défaveur de l'algorithme de recherche aléatoire. En effet, la complexité de l'espace d'action augmentant, la densité des actions tirées uniformément est moins dense, affaiblissant l'effort d'exploration de l'algorithme de recherche aléatoire.

Le problème de l'acrobot

Le problème de l'acrobot est identique à celui présenté dans le chapitre précédent — voir page 69 — à l'exception de l'espace d'action A qui est continu et défini par le domaine $[-2, 2]$.

L'expérimentation sur le problème du cart-pole consiste à calculer la moyenne sur 1000 états initiaux et pour différentes longueurs de séquences $H \in \{5, 10, 20, 40, 80, 100\}$ et différents nombres $n \in \{100, 1000, 10000\}$ d'appels au modèle génératif de la somme des récompenses sur 100 pas de temps. Les états initiaux sont tirés uniformément sur le domaine $[1, 5.28] \times [-1, 1] \times [1, 5.28] \times [-1, 1]$.

Nous pouvons observer sur le graphique 5.19, représentant la moyenne de la somme des récompenses obtenues pour $n = 100$ appels au modèle génératif, que les performances des trois variantes de l'algorithme de planification séquentielle sont quasi identiques et dominent presque l'algorithme de

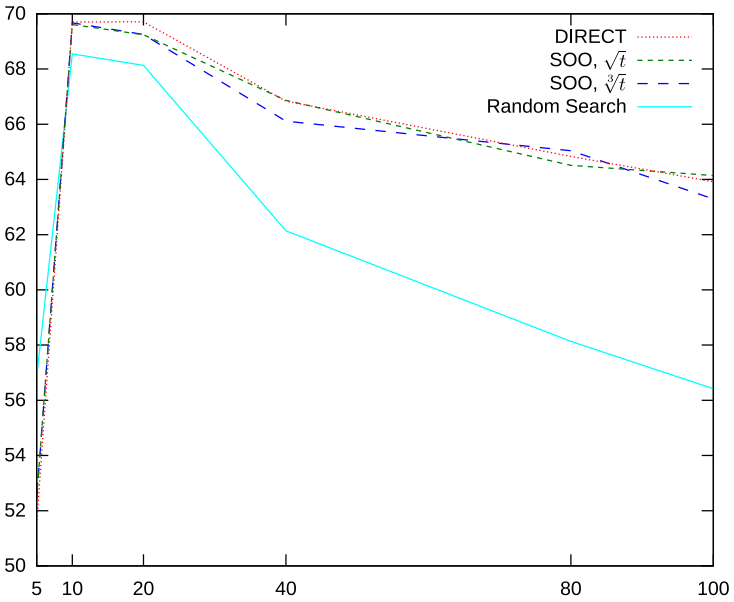


FIGURE 5.20 – Somme des récompenses pour le problème de l'acrobot avec $n = 1000$ appels au modèle génératif.

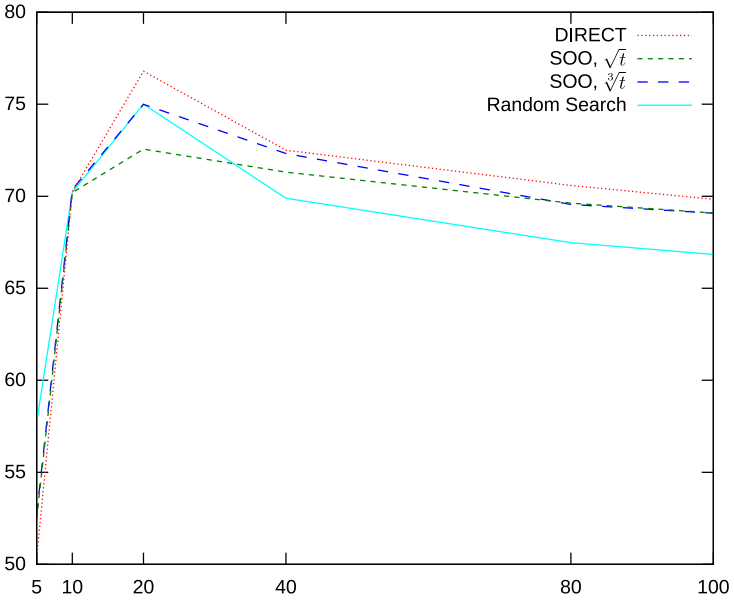


FIGURE 5.21 – Somme des récompenses pour le problème de l'acrobot avec $n = 10000$ appels au modèle génératif.

recherche aléatoire. L'algorithme de planification séquentielle utilisant l'algorithme SOO avec $h_{\max}(t) = \lfloor \sqrt{t} \rfloor$ obtient la meilleure performance pour $H = 10$. Ces bonne performances peuvent être expliqué par le fait que ma fonction récompense du problème de l'acrobot n'est pas discontinue — contrairement à celle des problèmes du cart-pole et du double cart-pole. Ainsi les algorithmes d'optimisation globale voient leur tâche facilitée.

Le graphique 5.20 représente la moyenne de la somme des récompenses obtenues pour $n = 1000$ appels au modèle génératif. Le nombre d'appels au modèle génératif augmentant, les différences dans la discrétisation se font plus apparentes. L'algorithme de planification séquentielle utilisant l'algorithme DIRECT obtient la meilleure performance pour $H = 20$. Il est à noter que l'algorithme de recherche aléatoire est de nouveau presque dominé par les trois variantes de l'algorithme de planification séquentielle.

Le graphique 5.21 représente la moyenne de la somme des récompenses obtenues pour $n = 10000$ appels au modèle génératif. De nouveau, la variante utilisant l'algorithme DIRECT obtient la meilleure performance pour $H = 20$. Il est intéressant d'observer que la différence de performance lorsque la longueur des séquences d'actions H passe de 10 à 20 est plus importante ici que pour les problèmes précédents alors que les performances obtenues pour $H = 10$ restent dans le même temps quasiment identique. Ce comportement peut être expliqué par l'efficacité accrue dans la discrétisation de l'espace d'action continu de par la régularité de la fonction récompense.

Le problème du bateau

Le problème du bateau est identique à celui présenté dans la section précédente — voir page 98.

L'expérimentation sur le problème du cart-pole consiste à calculer la moyenne sur 1000 états initiaux et pour différentes longueurs de séquences $H \in \{5, 10, 20, 40, 80, 100\}$ et différents nombres $n \in \{100, 1000, 10000\}$ d'appels au modèle génératif de la somme des récompenses sur 200 pas de temps. Les états initiaux sont tirés uniformément sur le domaine $[0, 0] \times [0, 200] \times [0, 0] \times [-1.57, 1.57] \times [0, 0] \times [0, 0] \times [0, 0]$.

Nous pouvons observer sur le graphique 5.22, représentant la moyenne de la somme des récompenses pour $n = 100$ appels au modèle génératif, que les trois variantes de l'algorithme de planification séquentielle obtiennent des performances supérieures à celle de l'algorithme recherche aléatoire. Ceci est dû à la nature très directrice de la fonction récompense, nature qui ne peut pas être pris en compte par l'algorithme de recherche aléatoire. L'algorithme de planification séquentielle utilisant l'algorithme DIRECT obtient la meilleure performance pour $H = 20$.

Le graphique 5.23 représente la moyenne de la somme des récompenses pour $n = 1000$ appels au modèle génératif. De nouveau, les trois variantes de l'algorithme de planification séquentielle obtiennent les meilleures perfor-

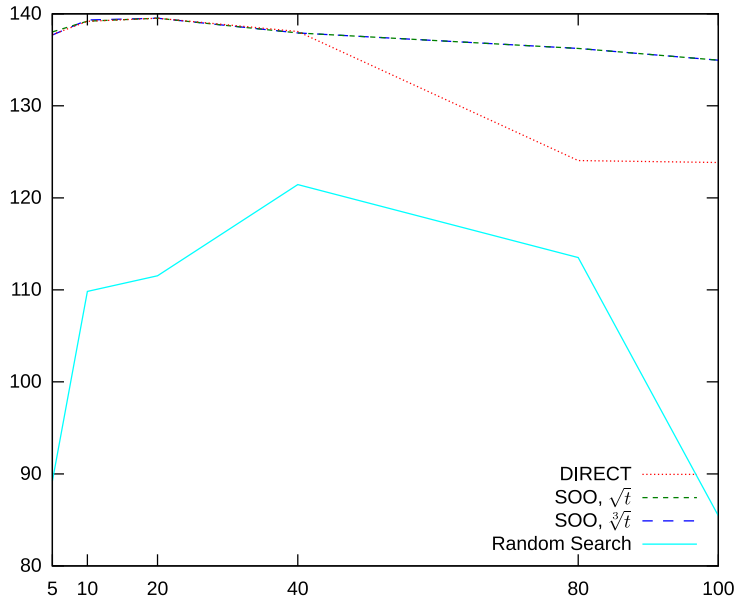


FIGURE 5.22 – Somme des récompenses pour le problème du bateau avec $n = 100$ appels au modèle génératif.

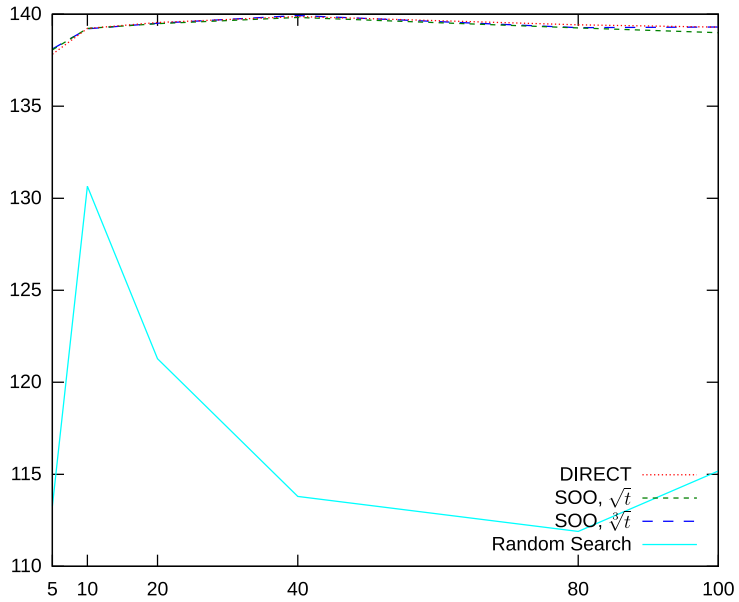


FIGURE 5.23 – Somme des récompenses pour le problème du bateau avec $n = 1000$ appels au modèle génératif.

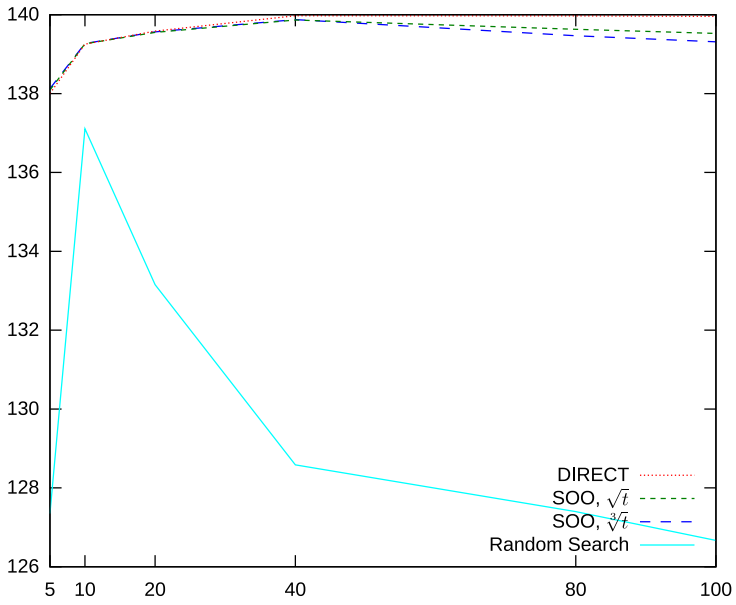


FIGURE 5.24 – Somme des récompenses pour le problème du bateau avec $n = 10000$ appels au modèle génératif.

mances avec un maximum atteint pour $H = 40$. Comme pour le problème de l'acrobot, les propriétés avantageuses de la fonction récompense permettent à l'algorithme de planification séquentielle d'atteindre de meilleures performances pour une longueur des séquences plus élevée que précédemment.

Le graphique 5.24 représente la moyenne de la somme des récompenses pour $n = 10000$ appels au modèle génératif. L'algorithme de recherche aléatoire est toujours dominé par les trois variantes de l'algorithme de planification séquentielle dont celle utilisant l'algorithme DIRECT obtient la meilleure performance pour $H = 40$. De plus, et contrairement à l'algorithme de planification séquentielle, les performances de l'algorithme de recherche aléatoire diminue fortement lorsque la longueur des séquences H augmente dans le cas du problème du bateau.

Le problème de la lévitation magnétique d'une boule en acier

Le problème de la lévitation magnétique d'une boule en acier est identique à celui présenté dans la section précédente — voir page 100.

L'expérimentation sur le problème de la lévitation magnétique d'une boule en acier consiste à calculer la moyenne de la somme des récompenses obtenues pour 1000 objectifs de position à atteindre par la boule en acier pour différentes longueurs de séquence $H \in \{5, 10, 20, 40, 80, 100\}$ et différents nombres $n \in \{100, 1000, 10000, 100000\}$ d'appels au modèle génératif. L'objectif de position change tous les 80 pas de temps. Les objectifs de posi-

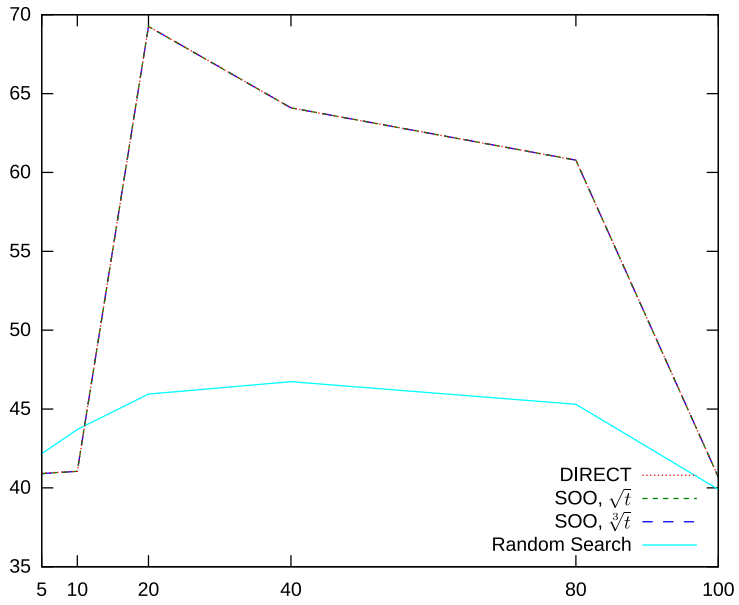


FIGURE 5.25 – Somme des récompenses pour le problème de la lévitation magnétique d'une boule en acier avec $n = 100$ appels au modèle génératif.

tion suivants ne sont pas connus à l'avance — c'est un paramètre du modèle génératif changé tout les 80 pas de temps. L'état initial est obtenu en appliquant l'action correspondant à un courant de 15V pendant 125 pas de temps en partant de l'état $(0.013, 0, 0)$ correspondant à l'état de repos de la boule en acier. Le premier objectif de position est alors effectif pour 80 pas de temps. Une fois ces 80 pas de temps écoulés, l'objectif de position est changé en conservant l'état courant du système — autrement dit, nous ne repartons pas de l'état initial. Les objectifs de position sont tirés uniformément sur le domaine $[0, 0.013]$.

Nous pouvons observer sur le graphique 5.25, représentant la moyenne de la somme des récompenses obtenues pour $n = 100$ appels au modèle génératif, que les performances des trois variantes de l'algorithme de planification séquentielle sont quasi identiques et trouvent leurs maximums pour $H = 20$. Les performance de l'algorithme de rechercher aléatoire sont en net retrait mais ne sont cependant pas complètement dominées.

Le graphique 5.26 représente la moyenne de la somme des récompenses obtenues pour $n = 1000$ appels au modèle génératif. L'algorithme de planification utilisant l'algorithme DIRECT obtient la meilleure performance pour $H = 10$ mais globalement, les performances des trois variantes sont très proches l'une de l'autre.

Le graphique 5.27 représente la moyenne de la somme des récompenses obtenues pour $n = 10000$ appels au modèle génératif. Comme précédemment,

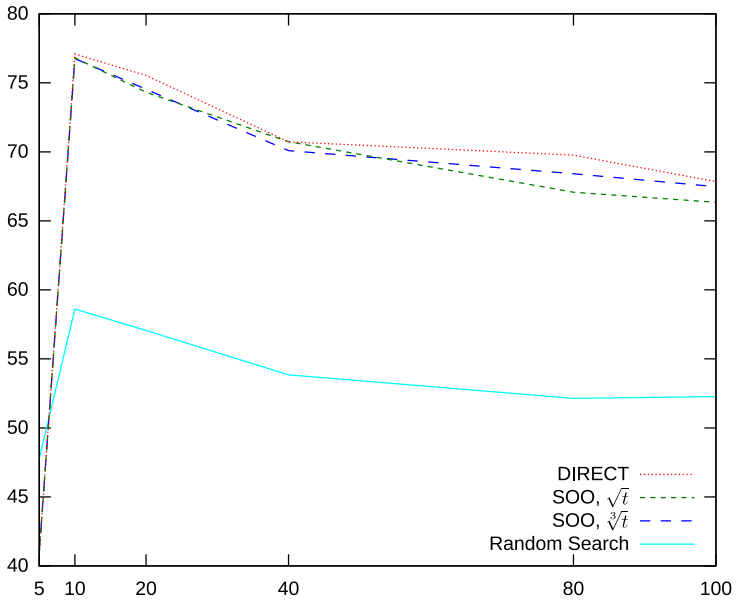


FIGURE 5.26 – Somme des récompenses pour le problème de la lévitation magnétique d'une boule en acier avec $n = 1000$ appels au modèle génératif.

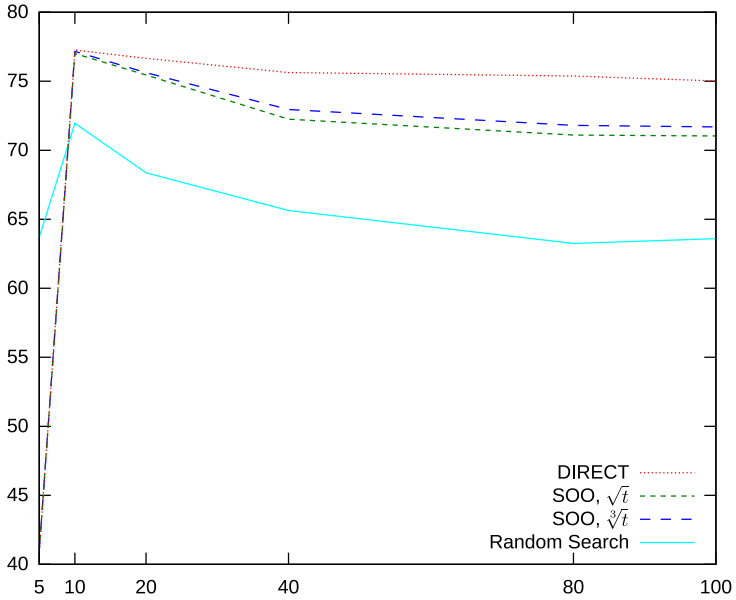


FIGURE 5.27 – Somme des récompenses pour le problème de la lévitation magnétique d'une boule en acier avec $n = 10000$ appels au modèle génératif.

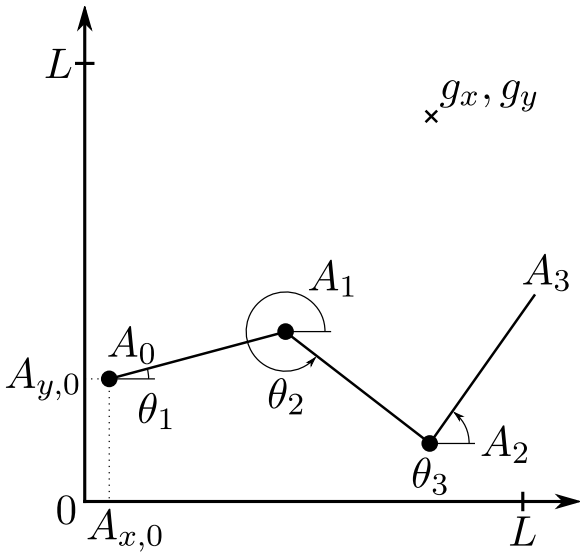


FIGURE 5.28 – Le problème du nageur à $s = 3$ segments.

la variante utilisant l'algorithme DIRECT obtient la meilleure performance pour $H = 10$ et se détache pour les longueurs de séquences plus élevées des deux autres variantes. L'algorithme de recherche aléatoire est encore en retrait par rapport aux trois variantes de l'algorithme de planification séquentielle.

Le problème du nageur

Le problème du nageur consiste à faire évoluer dans un milieu visqueux un nageur constitué de s segments relié par $s-1$ articulations sur lesquelles un couple peut-être appliqué — voir la figure 5.28. L'espace d'action est donc de dimension $s-1$. L'objectif est de faire atteindre une position fixe au centre de gravité du nageur. Les dynamiques du système — provenant de [Coulom, 2002] — sont décrites le système d'équations suivant avec $i \in \{1, \ldots, s\}$:

$$\begin{cases} \overrightarrow{f_s} = \overrightarrow{0} \\ m_i \frac{l_i}{12} \ddot{\theta}_i = \det(\overrightarrow{G_i A_i}, \overrightarrow{f_i} + \overrightarrow{f_{i-1}}) + M_i - a_i + a_{i-1} \end{cases}$$

avec pour $i \in \{1, \ldots, s\}$

$$\begin{cases} \overrightarrow{f_0} = \overrightarrow{0} \\ \overrightarrow{f_i} = \overrightarrow{f_{i-1}} + \overrightarrow{F_i} + m_i \ddot{G}_i \end{cases}$$

la force appliqué par le segment $i+1$ au segment i,

$$M_i = -k\dot{\theta}_i \frac{l_i^3}{12}$$

le moment total au centre de gravité G_i,

$$\overrightarrow{F_i} = -kl_i(\dot{G}_i \cdot \overrightarrow{n_i})\overrightarrow{n_i}$$

la force totale du segment i, $G_i = (A_{i-1} + A_i)/2$ le centre de gravité du segment i, $\overrightarrow{n_i}$ le vecteur normal au segment i, $l_i = 1$ longueur du segment i, $m_i = 1$ masse du segment i, θ_i l'angle du segment i par rapport au plan horizontal, A_{i-1} et A_i les coordonnées des deux extrémités du segment i, $a_i \in A$ le couple appliqué à l'articulation entre le segment i et le segment $i+1$ avec $A = [-5, 5] \times \ldots [-5, 5]$ l'espace d'action de dimension $s-1$ — $a_0 = 0$ et $a_s = 0$ — et $k = 10$ le coefficient de viscosité.

L'état du système à l'instant t est représenté par $6+2s$-uplet $(A_{x,0}, A_{y,0}, G_x, G_y, \dot{G}_x, \dot{G}_y, \theta_1, \dot{\theta}_1, \ldots, \theta_s, \dot{\theta}_s)$ avec G_x et G_y les coordonnées du centre de gravité du nageur. La fonction récompense est différente de celle définie par [Coulom, 2002] pour que les récompenses soient entre 0 et 1. Elle est définie pour un état $x_t = (A_{x,0,t}, A_{y,0,t}, G_{x,t}, G_{y,t}, \dot{G}_{x,t}, \dot{G}_{y,t}, \theta_{1,t}, \dot{\theta}_{1,t}, \ldots, \theta_{s,t}, \dot{\theta}_{s,t})$ et une action $a_t = (a_{1,t}, \ldots, a_{s-1})$ par l'équation :

$$r(x_t, a_t) = \max\left(0, \min\left(1, \frac{\sqrt{(G_{x,t+1} - g_x)^2 + (G_{y,t+1} - g_y)^2}}{L\sqrt{2}}\right)\right)$$

avec $(g_x, g_y) = (2.5, 4)$ les coordonnées de l'objectif et $L = 5$ la taille d'un côté du carré sur lequel évolue le nageur. Le facteur de dépréciation est de 0.95 et le pas de temps est de 0.02 avec 8 pas de temps intermédiaires.

L'expérimentation sur le problème du nageur consiste à calculer la moyenne sur 1000 états initiaux et pour différents nombre $s \in \{3, 4, 5, 6\}$ de segments, différentes longueurs de séquences $H \in \{5, 10, 20, 40, 80, 100\}$ et différents nombre $n \in \{100, 1000, 10000\}$ d'appels au modèle génératif de la somme des récompenses sur 300 pas de temps. Les états initiaux sont tirés uniformément sur le domaine $[0, 5] \times [0, 5] \times [-1, 1] \times [-1, 1] \times [-3.14, 3.14] \times [-1, 1] \times \ldots [-3.14, 3.14] \times [-1, 1]$. G_x et G_y sont initialisés en fonction de l'état tiré et doivent appartenir à $[0, 5]$. Dans le cas contraire, G_x et G_y sont modifiés ainsi que $A_{x,0}$ et $A_{y,0}$.

Nous pouvons observer sur le graphique 5.31, représentant la moyenne de la somme des récompenses obtenues pour $n = 100$ appels au modèle génératif pour un nageur à $s = 3$ segments, que l'algorithme de planification séquentielle utilisant l'algorithme DIRECT obtient la meilleure performance pour $H = 10$. Les deux variantes utilisant l'algorithme SOO ont des performances très proches et dépassent l'algorithme de recherche aléatoire pour $10 \leq H \leq 80$. Les trois variantes de l'algorithme de planification séquentielle ont des performances identiques pour $H \geq 40$ dû au faible nombre d'appels au modèle génératif alloués par rapport à la longueur des séquences.

Le graphique 5.30 représente la moyenne de la somme des récompenses obtenues pour $n = 1000$ appels au modèle génératif pour un nageur à $s =$

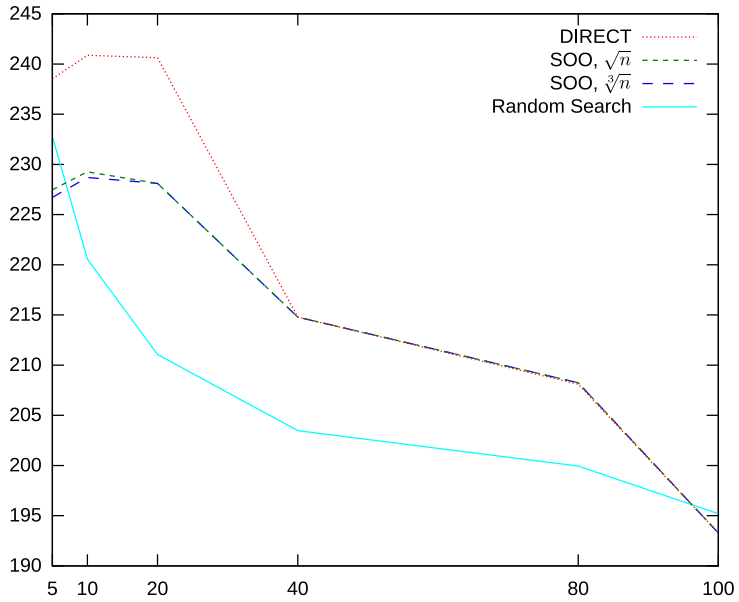


FIGURE 5.29 – Somme des récompenses pour le problème du nageur à $s = 3$ segments pour $n = 100$ appels au modèle génératif.

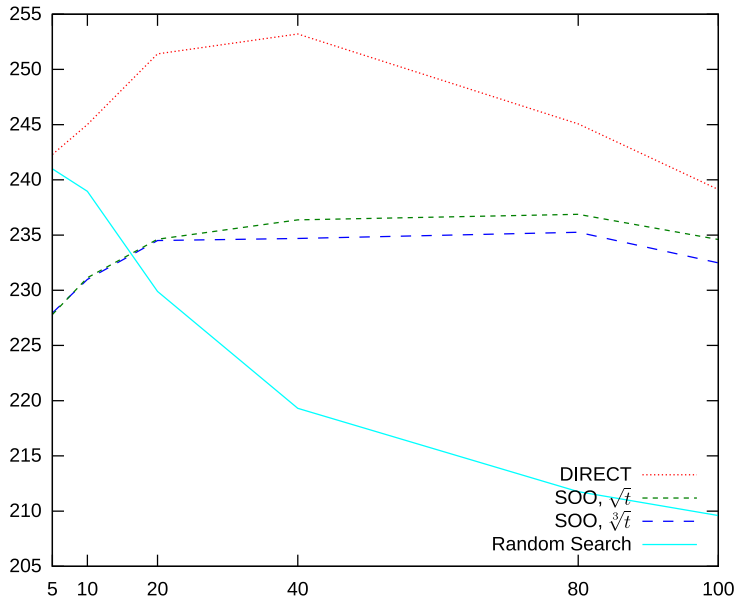


FIGURE 5.30 – Somme des récompenses pour le problème du nageur à $s = 3$ segments pour $n = 1000$ appels au modèle génératif.

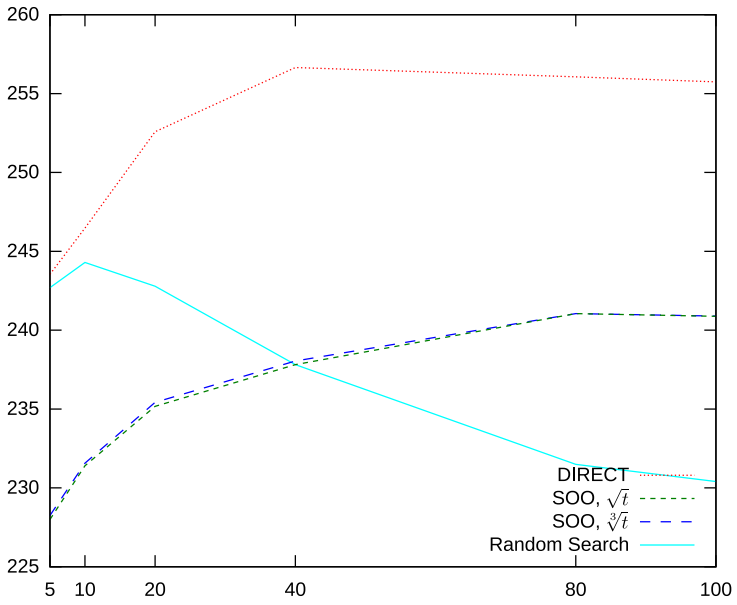


FIGURE 5.31 – Somme des récompenses pour le problème du nageur à $s = 3$ segments pour $n = 10000$ appels au modèle génératif.

3 segments. L'algorithme de planification séquentielle utilisant l'algorithme DIRECT domine les deux autres variantes ainsi que l'algorithme de recherche aléatoire et obtient la meilleure performance pour $H = 40$. Les deux variantes utilisant l'algorithme SOO commencent de manière identique mais pour $H \geq 40$, la variante utilisant l'algorithme SOO avec $h_{\max}(t) = \lfloor \sqrt{t} \rfloor$ se détache légèrement en tête. On peut noter que lorsque la longueur des séquences augmente, les performances des trois variantes de l'algorithme de recherche séquentielle restent stable.

Le graphique 5.31 représente la moyenne de la somme des récompense obtenues pour $n = 10000$ appels au modèle génératif pour un nageur à $s = 3$ segments. De nouveau, la variante de l'algorithme de planification séquentielle utilisant l'algorithme DIRECT domine les autres et obtient la meilleure performance pour $H = 40$. Les deux variantes utilisant l'algorithme SOO obtiennent quasiment les mêmes performances qui ne commencent à baisser que pour $H = 100$. L'algorithme de recherche aléatoire passe en dernière position pour $H \geq 80$.

Nous pouvons observer sur le graphique 5.32, représentant la moyenne de la somme des récompenses obtenues pour $n = 100$ appels au modèle génératif pour un nageur à $s = 4$ segments, que son allure général est très proche du graphique 5.29. L'algorithme de planification séquentielle utilisant l'algorithme DIRECT obtient la meilleure performance pour $H = 5$ et domine presque entièrement l'algorithme de recherche aléatoire. Nous pouvons ob-

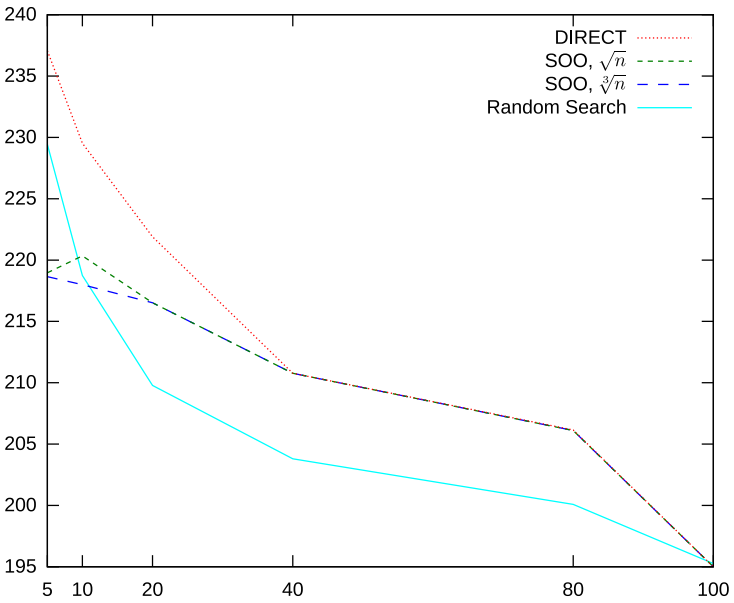


FIGURE 5.32 – Somme des récompenses pour le problème du nageur à $s = 4$ segments pour $n = 100$ appels au modèle génératif.

server que l'augmentation du nombre de dimension de l'espace d'action continu complique la tâche des algorithmes d'optimisation globale baissant du même coup la longueur des séquences obtenant la meilleure performance.

Le graphique 5.33 représente la moyenne de la somme des récompenses obtenues pour $n = 1000$ appels au modèle génératif pour un nageur à $s = 4$ segments. L'algorithme de planification utilisant l'algorithme DIRECT domine complètement l'algorithme de recherche aléatoire et obtient la meilleure performance pour $H = 40$. On peut observer cependant que pour $H \geq 80$, ses performances diminuent. Comme précédemment dans le cas où $s = 3$, les deux variantes utilisant l'algorithme SOO ont des performances très proches et qui sont en net retrait par rapport à celles de la variante utilisant l'algorithme DIRECT. Ce comportement est identique à celui observé pour le problème du double cart-pole où, pour rappel, la dimension de l'espace d'action continu était de 2.

Le graphique 5.34 représente la moyenne de la somme des récompenses obtenues pour $n = 10000$ appels au modèle génératif pour un nageur à $s = 4$ segments. L'algorithme de planification utilisant l'algorithme DIRECT obtient la meilleure performance pour $H = 100$. Nous pouvons observer que l'algorithme de recherche aléatoire domine les deux variantes de l'algorithme de planification séquentielle utilisant l'algorithme SOO. Il est à noter que, contrairement aux précédentes expérimentations sur le nageur, les performances de l'algorithme de planification séquentielle utilisant l'algorithme

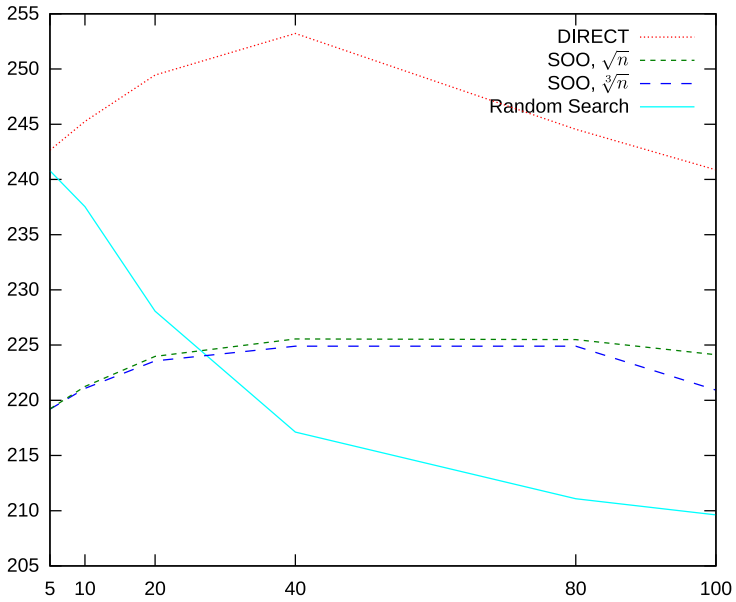


FIGURE 5.33 – Somme des récompenses pour le problème du nageur à $s = 4$ segments pour $n = 1000$ appels au modèle génératif.

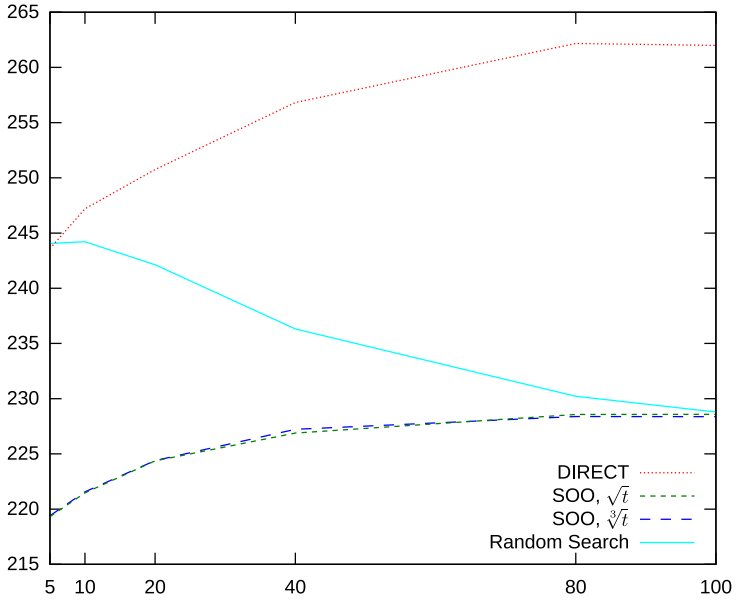


FIGURE 5.34 – Somme des récompenses pour le problème du nageur à $s = 4$ segments pour $n = 10000$ appels au modèle génératif.

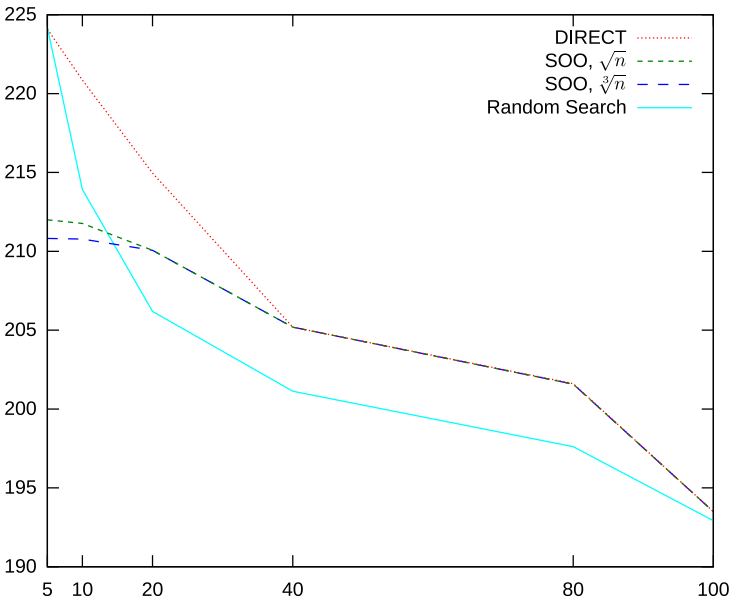


FIGURE 5.35 – Somme des récompenses pour le problème du nageur à $s = 5$ segments pour $n = 100$ appels au modèle génératif.

DIRECT continuent d'augmenter pour toutes les longueurs de séquences utilisées.

Nous pouvons observer sur le graphique 5.35, représentant la moyenne de la somme des récompenses obtenues pour $n = 100$ appels au modèle génératif pour un nageur à $s = 5$ segments, que les performances de l'algorithme de recherche aléatoire et de l'algorithme de planification séquentielle utilisant l'algorithme DIRECT très proches pour $H = 5$ mais pour les autres longueurs de séquences, ce dernier obtient les meilleures performances. Pour $H \geq 20$, l'algorithme de recherche aléatoire est dominé par les trois variantes de l'algorithme de planification séquentielle.

Le graphique 5.36 représente la moyenne de la somme des récompenses obtenues pour $n = 1000$ appels au modèle génératif pour un nageur à $s = 5$ segments. L'algorithme de planification séquentielle utilisant l'algorithme DIRECT domine l'algorithme de recherche aléatoire ainsi que les deux variantes de l'algorithme de planification séquentielle utilisant l'algorithme SOO et obtient la meilleure performance pour $H = 20$.

Le graphique 5.37 représente la moyenne de la somme des récompenses obtenues pour $n = 10000$ appels au modèle génératif pour un nageur à $s = 5$ segments. Comme pour le nageur à $s = 4$ segments, l'algorithme de planification séquentielle utilisant l'algorithme DIRECT obtient la meilleure performance pour $H = 100$. Il domine l'algorithme de recherche aléatoire pour toutes les longueurs de séquences sauf pour $H = 5$ où il est devancé.

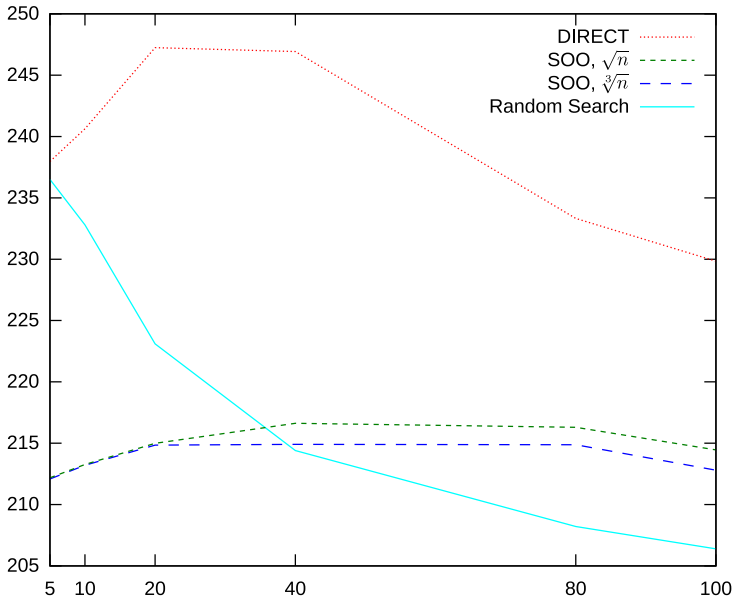


FIGURE 5.36 – Somme des récompenses pour le problème du nageur à $s = 5$ segments pour $n = 1000$ appels au modèle génératif.

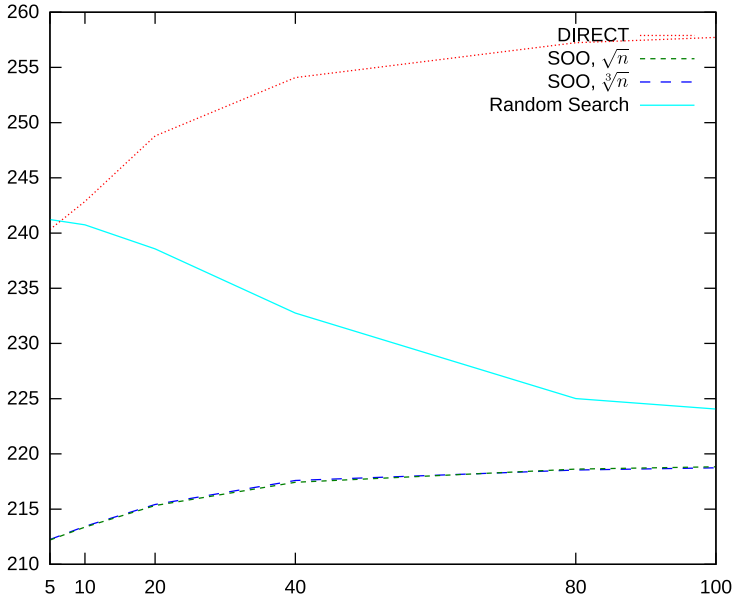


FIGURE 5.37 – Somme des récompenses pour le problème du nageur à $s = 5$ segments pour $n = 10000$ appels au modèle génératif.

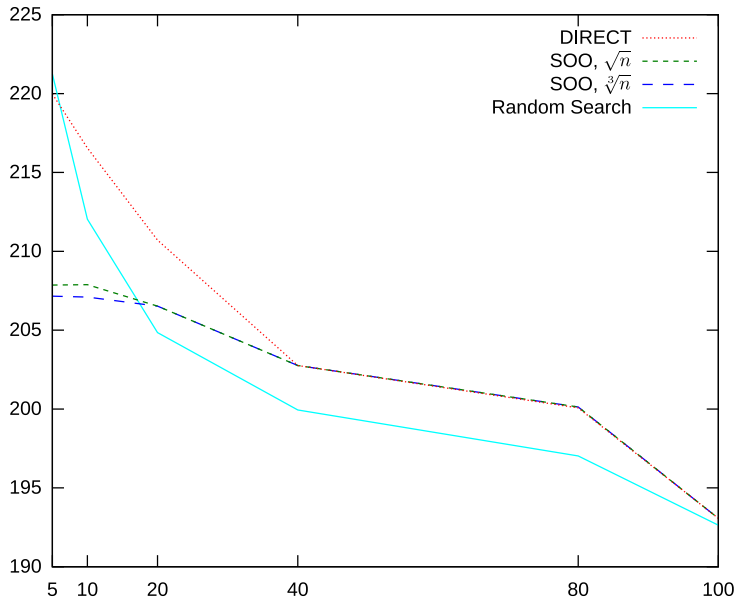


FIGURE 5.38 – Somme des récompenses pour le problème du nageur à $s = 6$ segments pour $n = 100$ appels au modèle génératif.

De plus, l'algorithme de recherche aléatoire domine les deux variantes de l'algorithme de planification séquentielle utilisant l'algorithme SOO.

Nous pouvons observer sur le graphique 5.38, représentant la moyenne de la somme des récompenses obtenues pour $n = 100$ appels au modèle génératif pour un nageur à $s = 6$ segments, que l'algorithme de recherche aléatoire obtient la meilleure performance pour $h = 5$ puis se fait dépasser pour les autres longueurs de séquences pour l'algorithme de planification séquentielle utilisant l'algorithme DIRECT. Comme précédemment, les trois variantes de l'algorithme de planification séquentielle obtiennent les mêmes performances pour $H \geq 40$.

Le graphique 5.39 représente la moyenne de la somme des récompenses obtenues pour $n = 1000$ appels au modèle génératif pour un nageur à $s = 6$ segments. Comme précédemment pour les différents nageurs et pour $n = 1000$ appels au modèle génératif, l'algorithme de recherche aléatoire est dominé par l'algorithme de planification séquentielle utilisant DIRECT. Celui-ci obtient la meilleure performance pour $H = 20$.

Le graphique 5.40 représente la moyenne de la somme des récompenses obtenues pour $n = 10000$ appels au modèle génératif pour un nageur à $s = 6$ segments. Nous pouvons observer que l'algorithme de planification séquentielle utilisant l'algorithme DIRECT obtient la meilleure performance $H = 80$ et domine presque l'algorithme de recherche aléatoire. Les deux variantes de l'algorithme de planification séquentielle utilisant l'algorithme

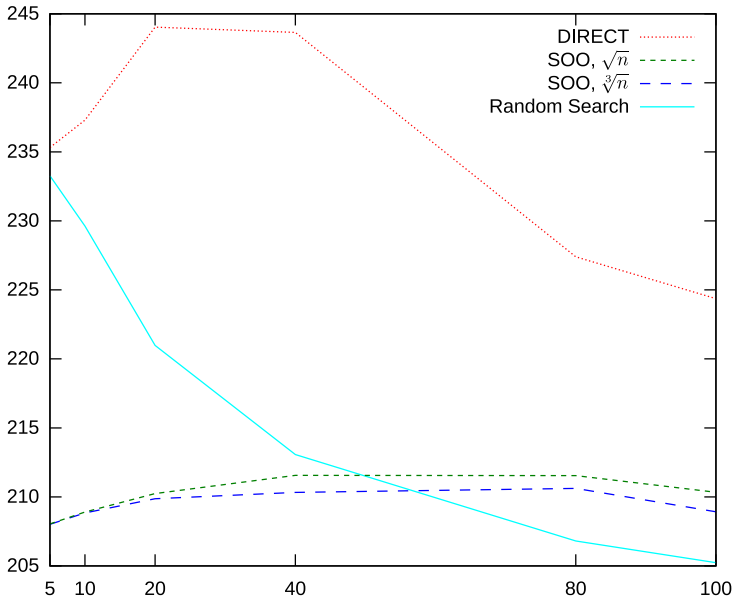


FIGURE 5.39 – Somme des récompenses pour le problème du nageur à $s = 6$ segments pour $n = 1000$ appels au modèle génératif.

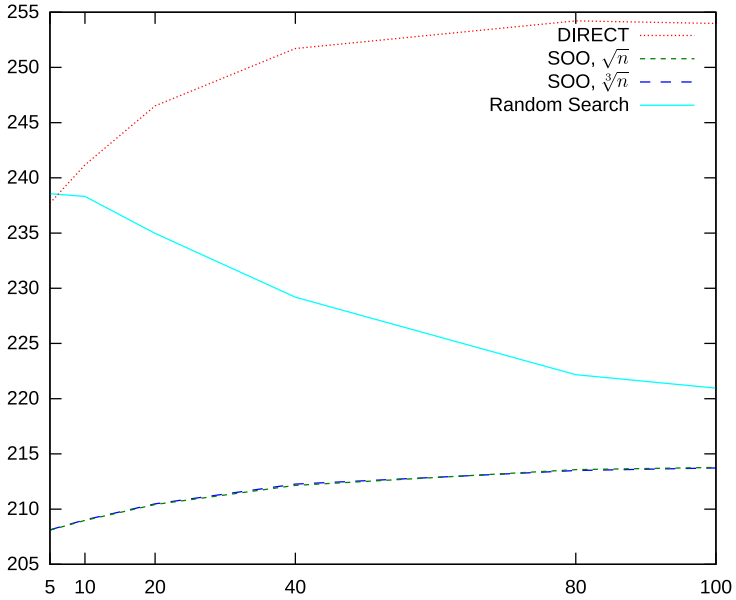


FIGURE 5.40 – Somme des récompenses pour le problème du nageur à $s = 6$ segments pour $n = 10000$ appels au modèle génératif.

SOO sont en net retrait.

Le problème du nageur et son nombre de dimensions variable pour l'espace d'action permet de mettre en évidence l'avantage de l'algorithme de planification séquentielle utilisant l'algorithme DIRECT vis-à-vis de l'algorithme de recherche aléatoire. Ceci peut s'expliquer par la baisse de la densité des points tirés par l'algorithme de recherche aléatoire dans l'espace d'action de par l'augmentation de son nombre de dimensions. Dans le cas de l'algorithme de planification séquentielle, les algorithmes d'optimisation globale comme l'algorithme DIRECT ont une efficacité accrue face à l'augmentation du nombre de dimensions, leur recherche étant structurée.

Nous pouvons conclure que l'algorithme de planification séquentielle obtient des résultats beaucoup plus probant que ceux obtenus précédemment avec l'algorithme de planification lipschitzienne et surtout supérieure à ceux obtenues par l'algorithme de recherche aléatoire. Cependant, le côté boucle ouverte de cette approche laisse à supposer que la politique obtenue ne peut être optimale vis-à-vis d'une politique en boucle fermée.

Chapitre 6

Conclusion

Au cours de cette thèse, nous avons présenté plusieurs contributions dans le contexte de la planification dans les processus de décision markovien et plus particulièrement dans le cas déterministe.

Nous avons décrit la planification, étant donné l'état courant du système à contrôler et un modèle génératif de celui-ci, comme une recherche avant pour trouver l'action ou la séquence d'actions à appliquer dans l'état courant. Ou d'une manière plus général, comme la définition en ligne d'une politique dont l'accès engendre un calcul non-trivial consistant à construire — dans une grande majorité des algorithmes — un arbre des possibilités, dont la racine est l'état courant du système à contrôler, grâce au modèle génératif.

Nous avons présenté différents algorithmes de la littérature ayant pour point commun leur application dans le cas stochastique. Ces algorithmes ont pour but de construire l'arbre des possibilités de manière intelligente et ce pour optimiser l'utilisation des ressources computationnelles disponibles. Nous avons aussi présenté l'approche optimiste présente dans la littérature et décrit comme une heuristique apportant une solution au dilemme exploration-exploitation et ainsi que ses différentes applications.

Notre première contribution est l'algorithme de planification optimiste qui s'inscrit dans le cadre de la planification déterministe avec espace d'action discret. Nous avons obtenu des résultats théoriques et, notamment, nous avons prouvé une borne supérieure sur le regret de l'algorithme de planification optimiste $R_{\mathcal{A}_O}(n) \leq \frac{\gamma^{d_n}}{1-\gamma}$ qui est à mettre en relation avec la borne supérieure sur le regret de l'algorithme de planification uniforme.

Nous avons aussi prouvé une borne supérieure sur le regret dans une classe particulière de problèmes où la proportion de chemins ϵ-optimaux est $O(\epsilon^\beta)$ avec $\beta \in [0, \frac{\log K}{\log 1/\gamma}]$. Cette borne est $R_{\mathcal{A}_O}(n) = O\left(n^{-\frac{\log 1/\gamma}{\log \kappa}}\right)$ avec $\kappa \stackrel{\text{def}}{=} K\gamma^\beta > 1$ ou $R_{\mathcal{A}_O} = O\left(\gamma^{\frac{(1-\gamma)^\beta n}{c'}}\right)$ lorsque $\kappa = 1$. Ces deux bornes nous montrent que les performances de l'algorithme de planification optimiste ne

peuvent pas être pire que celles de l'algorithme de planification uniforme et qu'elles sont mêmes potentiellement meilleures dans le cas où $\kappa < K$.

Pour vérifier cette affirmation et quantifier les performances de l'algorithme de planification optimiste, nous avons procédé à des expérimentations sur les problème de la balle, du cart-pole, de l'acrobot, du mountain car et du double cart-pole. Ces expérimentations nous ont montré que les performances de l'algorithme de planification optimiste étaient au pire identiques à celles de l'algorithme uniforme —les choix arbitraires étant effectués de manière identique pour les deux algorithmes – voir même en de nombreuses occasions nettement supérieures à celles de l'algorithme de planification uniforme. Cependant, les performances de l'algorithme de planification optimiste se sont révélées être sensibles au facteur de branchement, comme cela apparait notamment dans le cas du problème du double cart-pole, ce comportement étant normal au vu de la borne supérieure sur le regret.

Notre deuxième contribution est l'algorithme de planification lipschitzienne qui s'inscrit dans le cadre de la planification déterministe mais avec un espace d'action continu. Comme nous avons vu auparavant, que l'augmentation du facteur de branchement diminue les performances de l'algorithme de planification optimiste. Le passage à un espace d'action continu ne pouvait donc se faire avec une simple discrétisation uniforme en vu de créer un espace d'action discret, le facteur de branchement résultant étant trop élevé. De plus, une discrétisation uniforme ne prenant pas en compte les variations locales des dynamiques du système, nous avons envisagé une autre approche.

En posant comme hypothèse qu'il existe une régularité sur les récompenses telle que pour tout $t \geq 0, |r_t(\mathbf{a}) - r_t(\mathbf{a}')| \leq \sum_{i=0}^{t} L^{t-s+1}\|a_i - a'_i\|$ avec $\mathbf{a}$ et $\mathbf{a}' \in A^\infty$ des séquences infinies d'actions, la fonction transition et la fonction récompense doivent être lipschitziennes pour vérifier l'hypothèse. De là, nous avons dérivé une borne B_i sur A_i, un ensemble de sous-espace de A^∞ telle que :

$$B_i = \sum_{t=0}^{T_i} \gamma^t \left(r_t\left(\mathbf{a}_c{}^i\right) + \sum_{j=0}^{t} L^{t-j+1}\delta_j^i \right) + \frac{\gamma^{T_i+1}}{1-\gamma}$$

avec T_i le plus grand entier $t \leq n_i$ tel que $r_t(\mathbf{a}_c{}^i) + \sum_{j=0}^{t} L^{t-j+1}\delta_j^i \leq 1$. De cette borne, nous avons élaborer l'algorithme de planification lipschitzienne permettant de discrétiser l'espace des séquences infinies d'actions continu en sélectionnant de manière optimiste le sous-ensemble à explorer.

Nous avons effectué des expérimentations pour valider l'approche sur les problèmes du cart-pole, du double cart-pole, de l'acrobot, du bateau et de la lévitation magnétique d'une boule en acier. Les résultats nous ont montré qu'il était difficile d'évaluer à priori la constante de Lipschitz d'un problème donné. De plus, celle-ci devant être une borne supérieure, nous ne pouvons avoir une adaptation au paysage local issu des régularités des récompenses.

Notre troisième contribution est l'algorithme de planification séquentielle qui s'inscrit dans le même cadre que la précédente contribution. Cependant, l'approche envisagée est intuitive contrairement à l'approche précédente plus théorique et basée sur une hypothèse. Cette intuition est à mettre en relation avec l'algorithme HOOT et l'algorithme OLOP. Ainsi plutôt que de construire un arbre des possibilités, nous construisons des séquences d'actions, chaque action étant issue d'une instance d'un algorithme d'optimisation globale. Les algorithmes d'optimisation globale que nous avons utilisé sont l'algorithme DIRECT et l'algorithme SOO, tout deux prenant en compte la régularité de la fonction à optimiser sans demander la connaissance de cette régularité. Cette intuition peut être aussi mis en relation avec l'algorithme de recherche aléatoire où plutôt que de tirer les actions au hasard, nous utilisons un algorithme d'optimisation globale pour la sélection des actions formant une séquence. Cependant, et contrairement au deux contribution précédentes, l'algorithme de planification séquentielle ne possède pas de trait optimiste propre mais seulement au travers des algorithmes d'optimisation globale qu'il utilise comme l'algorithme SOO.

Nous avons effectué des expérimentations pour vérifier l'approche sur les problèmes du cart-pole, du double cart-pole, de l'acrobot, du bateau, de la lévitation magnétique d'une boule en acier et du nageur. Les résultats obtenus sont encourageant et dépassent en de nombreuses occasions ceux de l'algorithme de recherche aléatoire.

Une extension naturelle pour l'algorithme de planification séquentielle serait de lui adjoindre le procédé de l'Iterative-Deepening basé sur les ressources computationnelles déjà consommées ou bien des indicateurs provenant de l'algorithme d'optimisation globale utilisé — finesse de la discrétisation par exemple.

Cependant, et contrairement à l'algorithme de planification optimiste, il est difficile de réutiliser les informations obtenues au travers des appels au modèle génératif après chaque pas de temps du système à contrôler. Ceci est dû principalement au fait que chaque action n'apparait qu'une fois pour chaque position dans une séquence d'actions. Ainsi, l'action retournée par l'algorithme de planification séquentielle n'est en première position que pour une seule séquence d'actions limitant à $H - 1$ le nombre d'appels au modèle génératif pouvant être économisés. Il serait alors intéressant d'envisager l'utilisation d'algorithmes d'optimisation globale adaptés ou modifiés pour prendre en compte ce facteur.

La réutilisation d'un sous-arbre de l'arbre étendu par l'algorithme de planification optimiste dans le pas de temps suivant du système à contrôler est une optimisation non négligeable en terme d'économie de ressources computationnelles et d'autant plus lorsque κ est proche de 1. En effet, plus la proportion de chemins ϵ-optimaux est faible, plus le sous-arbre conservé d'un pas de temps à l'autre est grand. Il serait intéressant d'exprimer une borne sur le regret en prenant en compte ce facteur ainsi que chiffrer l'économie de

ressources computationnelles au cours d'expérimentations.

Dans le cadre de l'algorithme de planification optimiste et des diverses optimisations possibles, la parallélisation fut évoquée pour améliorer les performances lorsque les ressources computationnelles sont exprimées en terme de temps disponible. En particulier, la deuxième méthode développée se propose d'effectuer à l'avance et en parallèle les appels au modèle génératif et ce grâce à la liste des feuilles triée par leur borne associée. Ce principe pourrait être aussi réutilisé dans le cas d'un système embarqué possédant de la mémoire de stockage en quantité limitée et hétérogène dans sa vitesse d'accès. Ainsi, et en fonction du type de mémoire, les données nécessaires à l'algorithme de planification optimiste pourrait être réparties par rapport à leur borne. Ainsi les feuilles possédants des bornes faibles — et donc en queue de liste — seraient sur un support mémoire lent tandis que les feuilles possédants une borne élevée seraient sur un support mémoire rapide.

Enfin l'une des extensions les plus intéressantes est celle au cas stochastique. L'algorithme OPSS et l'algorithme OLOP utilisent une approche optimiste et s'appliquent au cas stochastique. Cependant, il serait intéressant d'envisager une solution dans le cas stochastique avec espace d'action continu. Une première piste serait de combiner l'algorithme OPSS ou l'algorithme OLOP avec l'approche décrite par [Pazis and Lagoudakis, 2009a] pour transformer un espace d'action continu en espace d'action discret. Une autre piste serait de reprendre l'algorithme de planification séquentielle et de l'adapter pour qu'il puisse utiliser un algorithme d'optimisation globale prenant en compte la stochasticité.

Bibliographie

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47 :235–256, 2002.

Leemon C. Baird and A. Harry Klopf. Reinforcement learning with high-dimensional continuous actions. Technical report, Wright Laboratory, Wright-Patterson Air Force Base, 1993.

Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artif. Intell.*, 72(1-2) :81–138, 1995.

Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

Amine Bourki, Guillaume Chaslot, Matthieu Coulm, Vincent Danjean, Hassen Doghmen, Jean-Baptiste Hoock, Thomas Hérault, Arpad Rimmel, Fabien Teytaud, Olivier Teytaud, Paul Vayssière, and Ziqin Yut. Scalability and parallelization of monte-carlo tree search. In *Computers and Games*, pages 48–58, 2010.

Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning : Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11 :1–94, 1999.

Steven J. Bradtke and Michael O. Duff. Reinforcement learning methods for continuous-time markov decision problems. In *Advances in Neural Information Processing Systems*, pages 393–400. MIT Press, 1994.

Ronen I. Brafman and Moshe Tennenholtz. R-max a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Leaning Research*, 3 :213–231, 2001.

Sébastien Bubeck and Rémi Munos. Open loop optimistic planning. In *Conference on Learning Theory*, 2010.

Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. Online optimization in x-armed bandits. In *NIPS*, pages 201–208, 2008.

Lucian Buşoniu, Rémi Munos, Bart De Schutter, and Robert Babuška. Optimistic planning for sparsely stochastic systems. In *Proceedings of the 2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL 2011)*, pages 48–55, 2011.

Tristan Cazenave and Nicolas Jouandeau. On the parallelization of uct. In *CGW 2007*, pages 93–101, June 2007.

Hyeong Soo Chang, Michael C. Fu, Jiaqiao Hu, and Steven I. Marcus. An adaptive sampling algorithm for solving markov decision processes. *Oper. Res.*, 53 :126–139, 2005.

Guillaume Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel monte-carlo tree search. In *Computers and Games*, pages 60–71, 2008.

Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. *CoRR*, 2007.

Rémi Coulom. *Reinforcement Learning Using Neural Networks with Applications to Motor Control*. PhD thesis, Institut National Polytechnique de Grenoble, 2002.

Scott Davies, Andrew Y. Ng, and Andrew Moore. Applying online search techniques to continuous-state reinforcement learning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 753–760, 1998.

Damien Ernst, Pierre Geurts, Louis Wehenkel, and Michael L. Littman. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6 :503–556, 2005.

Chris Gaskett, David Wettergreen, and Alexander Zelinsky. Q-learning in continuous state and action spaces. In *In Australian Joint Conference on Artificial Intelligence*, pages 417–428. Springer-Verlag, 1999.

Sylvain Gelly and Yizao Wang. Exploration exploitation in go : Uct for monte-carlo go. In *In Twentieth Annual Conference on Neural Information Processing Systems (NIPS)*, 2006.

Sylvain Gelly, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, and Yann Kalemkarian. The parallelization of monte-carlo planning. In *ICINCO*, 2008.

Mohammad Ghavamzadeh and Sridhar Mahadevan. Continuous-time hierarchical reinforcement learning. In *In Proceedings of the Eighteenth International Conference on Machine Learning*, pages 186–193. Morgan Kaufmann, 2001.

Roland Hafner and Martin Riedmiller. Reinforcement learning in feedback control - challenges and benchmarks from technical process control. *Machine Learning*, 84(1-2) :137–169, 2011.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. In *Systems Science and Cybernetics, IEEE Transactions on*, volume 4, pages 100–107, 1968.

Ronald A. Howard. *Dynamic Programming and Markov Processes*. The MIT Press, Cambridge, MA, 1960.

Jean-François Hren and Rémi Munos. *Optimistic Planning of Deterministic Systems*, pages 151–164. Springer-Verlag, 2008.

Thomas Jaksch, Ronald Ortner, and Peter Auer. Near-optimal regret bounds for reinforcement learning. *J. Mach. Learn. Res.*, 99 :1563–1600, 2010.

Donald R. Jones, Cary D. Perttunen, and Bruce E. Stuckman. Lipschitzian optimization without the lipschitz constant. *Journal of Optimization Theory and Application*, 79 :157–181, October 1993.

Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. *Mach. Learn.*, 49 :193–208, 2002.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, pages 282–293, 2006.

Sven Koenig and Maxim Likhachev. Real-time adaptive a*. In *AAMAS*, pages 281–288, 2006.

Sven Koenig, Maxim Likhache, Yaxin Liu, and David Furcy. Incremental heuristic search in ai. *Artificial Intelligence Magazine*, 25(2) :99–112, 2004a.

Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning a*. *Artif. Intell.*, 155(1-2) :93–146, may 2004b.

Richard E. Korf. Depth-first iterative-deepening : An optimal admissible tree search. *Artif. Intell.*, 27(1) :97–109, 1985.

Tze Leung Lai and Herbert E. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6 :4–22, 1985.

Alessandro Lazaric, Marcello Restelli, and Andrea Bonarini. Reinforcement learning in continuous action spaces through sequential monte carlo methods. In *NIPS*, 2007.

Maxim Likhachev, Geoff Gordon, and Sebastian Thrun. ARA* : Anytime A* with provable bounds on sub-optimality. *Advances in Neural Information Processing Systems*, 16, 2004.

Francis Maes, Louis Wehenkel, and Damien Ernst. Optimized look-ahead tree policies. *European Workshop on Reinforcement Learning (EWRL'9)*, 2011.

Christopher R. Mansley, Ari Weinstein, and Michael L. Littman. Sample-based planning for continuous action markov decision processes. In *ICAPS*, 2011.

Nicolas Meuleau and Paul Bourgine. Exploration of multi-state environments : Local measures and back-propagation of uncertainty. *Machine Learning*, 35(2) :117–154, 1999.

José Del R. Millán, Daniele Posenato, and Eric Dedieu. Continuous-action q-learning. *Mach. Learn.*, 49(2-3) :247–265, 2002.

Rémi Munos. Optimistic optimization of deterministic functions without the knowledge of its smoothness. In *Advances in Neural Information Processing Systems*, 2011.

Gerhard Neumann, Michael Pfeiffer, and Wolfgang Maass. Efficient continuous-time reinforcement learning with adaptive state graphs. In *Proceedings of the 18th European conference on Machine Learning*, ECML '07, pages 250–261, 2007.

Jason Pazis and Michail G. Lagoudakis. Binary action search for learning continuous-action control policies. In *ICML*, page 100, 2009a.

Jason Pazis and Michail G. Lagoudakis. Learning continuous-action control policies. In *IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL 2009)*, pages 169–176, 2009b.

Laurent Péret and Frédérick Garcia. On-line search for solving markov decision processes via heuristic sampling. In *ECAI*, pages 530–534, 2004.

Martin L. Putterman. *Markov Decision Processes : Discrete Stochastic Dynamic Programming*. Wiley-Interscience, 1994.

Brian Sallans, Geoffrey E. Hinton, and Sridhar Mahadevan. Reinforcement learning with factored states and actions. *Journal of Machine Learning Research*, 5 :1063–1088, 2004.

Anthony Stentz. Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '94)*, volume 4, pages 3310–3317, 1994.

Anthony Stentz. The focussed d* algorithm for real-time replanning. In *In Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1652–1659, 1995.

Richard S. Sutton. Dyna, an integrated architecture for learning, planning and reacting. *SIGART Bulletin*, 2(4) :160–163, 1991.

Istvan Szita and András Lörincz. The many faces of optimism : a unifying approach. In *ICML*, pages 1048–1055, 2008.

Gerald Tesauro and Gregory R. Galperin. On-line policy improvement using monte-carlo search. In *Neural Information Processing Systems*, pages 1068–1074, 1996.

Christopher Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge,England, 1989.

Kazuki Yoshizoe, Akihiro Kishimoto, Tomoyuki Kaneko, Haruhiro Yoshimoto, and Yutaka Ishikawa. Scalable distributed monte-carlo tree search. In *SoCS'11*, pages 180–187, 2011.